User’s Guide

C6000 到 C7000 迁移

摘要

本文档提供的信息可帮助您将现有 C6000™ 源代码迁移到为 C7000™ (C7x) 处理器系列编写的代码。这样，该代码便可由 C7000 编译器编译。这个过程不需要额外的工具；只需要注意 C6000 和 C7000 处理器系列之间的编译器功能有何不同。

内容

English Document: SPRUIG5

1 关于本文档

应该对最初为 C6000 系列编写的源代码进行本文档中描述的更改，以便迁移到 C7000 (C7x) 处理器。具体而言，本文档描述了您需要评估和手动迁移的源代码的各个方面。

本文档还描述了 `c6x_migration.h` 头文件提供的支持，该文件包含在 C7000 运行时支持库中。您可以包含此文件以方便编译。对于没有对地址、控制寄存器或内存映射外设进行硬编码引用的应用程序，#include 这个头文件应该足以进行构建和运行。

本文档并非用作 C6000 或 C7000 编译器工具链的编译器用户指南。因此假定您熟悉 C6000 编译器。

C7000 编译器不支持两种主要的 C6000 编程范例，因此不在本文档中讨论：

- C6000 线性汇编
- C6000 手工编码汇编

以这些格式中的任何一种编写的现有 C6000 源代码都需要用 C 或 C7000 汇编语言重写，以便由 C7000 编译器编译。

1.1 相关文档

以下文档将提供 C7x 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)
- *C6000 C/C++ 优化编译器用户指南* (SPRUI04)
- *C7000 嵌入式应用程序二进制接口 (EABI) 参考指南* (SPRUIG4)
- *C7000 主机仿真用户指南* (SPRUIG6)
- *VCOP Kernel-C 至 C7000 迁移工具用户指南* (SPRUIG3)

1.2 商标

2 将 C 源代码从 C6000 迁移到 C7000

本章中的相关部分介绍了需要对源代码做出的更改以及 c6x_migration.h 头文件提供的支持。

2.1 编译器选项

将 C6000 代码移植到 C7000 时更改以下编译器命令行选项：

- 将 --silicon_version 选项设置为 --silicon_version=7100。（或将 -mv6600、-mv6740 或 -mv6400+ 选项替换为 -mv7100。）
- --interrupt_threshold (-mi) 选项将被忽略。C7000 C 代码始终可中断。
- --speculate_loads (-mh) 选项将被忽略。C7000 编译器对除 ioport 变量/地址之外的所有加载项使用推测加载指令。
- 使用--opt_for_speed (-mf) 选项指定大小/速度权衡选项。使用表 2-1 将以前使用的 --opt_for_space (-ms) 选项更改为相应的 -mf 选项。C7000 不存在 --opt_for_space 选项。即使没有为 C6000 使用 --opt_for_space 选项，也**必须使用** --opt_for_speed 选项。

表 2-1. 相应的 -ms 和 -mf 选项

--opt_for_space (-ms) 级别（仅限 C6000）	--opt_for_speed (-mf) 级别（C6000 和 C7000）	说明
无 C6000 选项	-mf5	C7000 上性能最佳；代码大小可能非常大
未选择 -ms 选项	-mf4	C7000 上接近最佳性能；C6000 上性能最佳
-ms0	-mf3	优化性能
-ms1	-mf2	优化代码大小
-ms2	-mf1	代码大小接近最小
-ms3	-mf0	最小代码大小

考虑使用 --opt_level=3 (-O3) 编译器选项。此选项可让编译器进行高级别的优化。

2.2 原生矢量数据类型

C6000 编译器支持使用原生矢量数据类型，这些类型在 *TMS320C6000 优化编译器用户指南* (SPRUI04) 的第 7.4.2 节中进行了说明。

将依赖原生矢量数据类型的源代码移植到 C7000 编译器时，请了解以下差异：

- 没有 c6x_vec.h 文件。将 c6x_vec.h 和 c6x.h 的 #include 替换为仅 c7x.h
- C7000 默认启用 --vectypes 编译器选项。
- 如果符号与原生矢量类型的名称相冲突，请使用 --vectypes=off。

在 C6000 上，必须使用 --vectypes 选项启用类似于 OpenCL™ 的原生矢量数据类型。但是，对于 C7000 编译器，默认情况下会启用所有原生矢量数据类型。因此，无需使用 --vectypes=on。如果现有代码的符号与原生矢量数据类型符号相冲突，则可以使用 --vectypes=off 选项关闭编译器对这些符号的识别。

请注意，每种原生矢量数据类型都有两个名称：一个不带双下划线前缀（例如 int4），一个带双下划线前缀（例如 __int4）。原生矢量数据类型的双下划线版本始终被编译器识别。--vectypes=off 选项仅会关闭那些没有双下划线前缀的矢量数据类型。

为了以后实现尽可能出色的兼容性和可移植性，我们建议您重命名使用 OpenCL 和类似 OpenCL 的原生矢量数据类型名称的所有现有 定义类型、结构或等级（不会成为原生矢量数据类型）。这也允许使用较短的原生矢量类型名称，这些名称不使用双下划线前缀。

C7100 和 C7120 的矢量大小为 512 位，因此矢量中的元素数上限大于 C6000。C6000 受到 OpenCL 施加的 16 个元素的限制。C7000 的矢量长度限于表 2-2 中所示的最大元素数。

表 2-2. C7000 支持的矢量类型

类型	说明	最大元素数
char*n*	由 *n* 个 8 位有符号整数值组成的矢量	64

表 2-2. C7000 支持的矢量类型 (continued)

类型	说明	最大元素数
uchar*n*	由 *n* 个 8 位无符号整数值组成的矢量	64
short*n*	由 *n* 个 16 位有符号整数值组成的矢量	32
ushort*n*	由 *n* 个 16 位无符号整数值组成的矢量	32
int*n*	由 *n* 个 32 位有符号整数值组成的矢量	16
uint*n*	由 *n* 个 32 位无符号整数值组成的矢量	16
long*n*	由 *n* 个 64 位有符号整数值组成的矢量	8
ulong*n*	由 *n* 个 64 位无符号整数值组成的矢量	8
float*n*	由 *n* 个 32 位单精度浮点值组成的矢量	16
double*n*	由 *n* 个 64 位双精度浮点值组成的矢量	8
cchar*n*	由 *n* 对 8 位有符号整数值组成的矢量	32
cshort*n*	由 *n* 对 16 位有符号整数值组成的矢量	16
cint*n*	由 *n* 对 32 位有符号整数值组成的矢量	8
clong*n*	由 *n* 对 64 位有符号整数值组成的矢量	4
cfloat*n*	由 *n* 对 32 位浮点值组成的矢量	8
cdouble*n*	由 *n* 对 64 位浮点值组成的矢量	4

C7000 不支持 C6000 64 位 longlong*n*、ulonglong*n* 和 clonglong*n* 矢量类型。只要包含 `c6x_migration.h` 文件，编译器就会将这些类型映射为 C7000 支持的相应类型，即分别为 long*n*、ulong*n* 和 clong*n*。

2.3 类型限定符：near 和 far

`near` 和 `far` 关键字不再需要，并将被 C7000 编译器忽略。我们建议您删除这些关键字的所有实例。

2.4 64 位 long 类型

C7000 上的 `long` 类型是 64 位，对应于 LP64 模型。

C6000 上的 `long` 类型为 32 位。

当不同机器和编译器之间实现可移植性需要特定的数据类型大小时，建议代码使用 C 标准整数类型 `int64_t` 和 `int32_t`（等）。这些标准整数类型在 `stdint.h` 中定义，后者是运行时支持库中包含的 C 标准库支持的一部分。

（ C6000 和 C7000 上的 `int` 类型为 32 位。 ）

2.5 对控制寄存器的引用

需要手动更改对 C 和 C++ 中控制寄存器的引用。C7000 有一组完全不同的控制寄存器。更多详细信息，请参阅 *C7000 CPU 和指令集参考指南*。

编译器工具支持的控制寄存器符号在 C6000 编译器工具的 `c6x.h` 和 C7000 编译器工具的 `c7x_cr.h` 和 `c7x_ecr.h` 中列出。在这些头文件中使用 `__cregister` 关键字来声明控制寄存器。

需要更改的代码的常见示例是：

- **对 C6000 控制状态寄存器 (CSR) 的引用。**这包括对饱和 (SAT) 位的引用。对于 C7000，这现在是标志状态寄存器 (FSR) 中的一个位。请注意，提供 SAT 位接口只是为了确保与 C6000 特定代码的兼容性。不推荐在编写新的 C7000 代码时引用 SAT 位。

  可以使用在 `c6x_migration.h` 中定义的 `__get_C7X_FSR()` API 来访问 SAT 位。返回一个 8 位值，其中 SAT 位指定为“位 7”。
- **对浮点配置寄存器 (FADCR、FAUCR、FMCR) 的引用。**与浮点运算有关的位现在是 C7000 标志状态寄存器 (FSR) 和浮点控制寄存器 (FPCR) 中的位。

  可以使用在 `c6x_migration.h` 中定义的 `__get_C7X_FSR()` API 来访问浮点状态位。返回一个 8 位值，其中浮点状态位包含位 0-6。

2.6 存储器映射外设

必须对存储器映射寄存器和外设的任何使用进行调查并根据需要进行更改。

- 检查相应 SDK 或 CSL (芯片支持库) 的文档以确定对外设的调用是否需要修改。
- 如果代码针对外设接口使用硬编码地址，则可能需要更改这些地址。
- 如果代码声明了指向外设控制寄存器的存储器映射指针，则必须确保在声明/定义指针变量时使用了 `volatile` 关键字。这样，编译器可以使用适当的存储器指令来访问存储器映射数据。 (编译器需要知道此信息，以便使用常规的非推测性负载。)

我们建议您为器件使用适当的 SDK 和 CSL 并查阅相关文档。

2.7 运行时支持

使用 C7000 编译器编译为 C6000 编写的代码时，必须在已迁移源文件的开头 #include (包含) C6000 至 C7000 迁移参考头文件 `c6x_migration.h`。

- 对于大多数应用，包含此头文件应该足以进行构建和运行。
- 此文件作为参考设计方案提供。您可以修改或重命名该文件。例如，将文件重命名为 `c6x.h` 将无需更改工程中的许多 #include 指令。
- 在 C6000 和 C7000 代码之间转换时，您可以 #include (包含) `c6x_migration.h` 和 `c7x.h`。
- 不依赖 C6000 内在函数的 C/C++ 源代码不需要迁移头文件。

如果要在迁移时删除对 C6000 的所有引用，则无需包含 C6000 到 C7000 的迁移引用头文件。反之，删除或修改对 C6000 特定内在函数和定义的所有引用。在这种情况下，将所有 `#include <c6x.h>` 实例替换为 `#include <c7x.h>`。

2.8 迁移头文件 c6x_migration.h 的内容

以下各节介绍了所提供的 c6x_migration.h 头文件的各个重要部分以及如何高效使用它。

2.8.1 支持的宏

c6x_migration.h 迁移头文件重新定义了由 C6000 编译器工具链在内部定义的宏。这些定义映射到相应的 C7000 定义。

- **C6000 目标宏：**以下所有目标宏均映射到 __C7000__。
  - __TMS320C6X__
  - _TMS320C6X
- **C6000 子目标宏：**目前，以下所有子目标宏均映射到 __C7100__。
  - _TMS320C6600
  - _TMS320C6740
  - _TMS320C6700_PLUS
  - _TMS320C67_PLUS
  - _TMS320C6700
  - _TMS320C64_PLUS
  - _TMS320C6400_PLUS
  - _TMS320C6400
- **字节序宏：**弃用了以下宏。C7000 编译器定义了应该使用的 __big_endian__ 或 __little_endian__。
  - _BIG_ENDIAN
  - _LITTLE_ENDIAN
- **EABI 宏：**弃用了以下宏。应改用 __TI_EABI__。
  - __TI_ELFABI__

2.8.2 不受支持的宏

以下宏不由 c6x_migration.h 迁移头文件定义。您应该更改代码或通过 --define (-D) 编译器选项手动定义宏。

- __DSBT__ — 不支持。
- __TI_TLS__ 和 __TI_USE_TLS — 尚未实现。
- __TI_32_BIT_LONG__ 和 __TI_40_BIT_LONG__ — 所有这些宏均未定义，因为 C7000 上的 long 始终为 64 位。
- _LARGE_MODEL、_SMALL_MODEL、_LARGE_MODEL_OPTION — 这些 C6000 宏依赖于 C7000 不再支持的各种内存模型选项。如果您的代码需要定义这些宏，请使用 --define (-D) 编译器选项。

2.8.3 传统数据类型

某些源文件可能会引用以下数据类型。

- __float2_t — 这是针对 2 个浮点值的“容器”。它在 C6000 和 C7000 中被定义为 double 类型。所有使用 __float2_t 的 C6000 内在函数都是在 c6x_migration.h 中声明的。
- __x128_t — 在 C6000 上，这是一个矢量“容器”类型，一个特殊的 128 位大小的结构矢量。在 C7000 上，此类型在 c6x_migration.h 中定义。所有使用 __x128_t 的 C6000 内在函数都是在 c6x_migration.h 中声明的。
- __int40_t — 在 C6000 上，这是特殊的一级整数类型，如 int 和 short。它具有 40 位精度。在 C6000 上，在原生运算（例如 +、- ）以及内在函数中使用此类型是有效的。C7000 没有定义此类型，不支持对应的运算。C7000 提供了 64 位值的 40 位饱和的内在函数 (VSATLW)。

2.8.4 传统内在函数

在 c6x.h 中定义的 C6000 内在函数名称与任何 C7x 内在函数名称并不冲突。因此，同时包含迁移头文件 c6x_migration.h 以及 c7x.h 不会导致出现问题。每个 C6000 内在函数都映射到单个 C7000 指令或一组执行或模拟相同行为的 C7000 指令。

- 如果 C6000 内在函数映射到单个 C7000 指令，则在 c7x.h 中搜索该指令的 C7000 C 习语。
- 如果 C6000 内在函数映射到一组指令，则 c7x.h 中将提供 C7000 C 习语示例。

例如，在 c6x_migration.h 中，声明了 _dadd 内在函数，并且在声明上方的注释中指示了映射的 C7000 指令 VADDW：

```
/* VADDW */
long long __BUILTIN _dadd(long long, long long);
```

在 c7x.h 中，显示了相同的指令及其支持的 C 习语，无论是 C 内在函数还是运算符：

```
VADDW
int = int + int;
int2 = int2 + int2;
int4 = int4 + int4;
int8 = int8 + int8;
int16 = int16 + int16;
cint = cint + cint;
cint2 = cint2 + cint2;
cint4 = cint4 + cint4;
cint8 = cint8 + cint8;
uint = uint + uint;
uint2 = uint2 + uint2;
uint4 = uint4 + uint4;
uint8 = uint8 + uint8;
uint16 = uint16 + uint16;
```

再举一例，声明了 _unpkbu4 内在函数，但没有与之对应的单个 C7000 指令。因此，c7x.h 显示的 C7000 C 习语如下所示：

```
/*--------------------------------------------------------------------------*/
/* _unpkbu4 使用 VUNPKLUB 和 VUNPKHUB 解包       */
/* 参数的低和高 2 个字节，然后构造结果。等效 C7X  */
/* 代码段如下所示：                                                         */
/*                                                                          */
/* ushort4 _unpkbu4(uchar4 src)                                             */
/* {                                                                        */
/*   ushort4 dst;                                                           */
/*   dst.lo = __unpack_low(src);                                            */
/*   dst.hi = __unpack_high(src);                                           */
/*   return dst;                                                            */
/* }                                                                        */
/*--------------------------------------------------------------------------*/
long long __BUILTIN _unpkbu4(unsigned)
```

C7000 编译器工具不支持以下已弃用的 C6000 内在函数。请改用 long long 变体：

- _mpy2 不受支持，请改用 _mpy2ll
- _mpyhi 不受支持，请改用 _mpyhill
- _mpyli 不受支持，请改用 _mpylill
- _mpyid 不受支持，请改用 _mpyidll
- _mpysu4 不受支持，请改用 _mpysu4ll
- _mpyu4 不受支持，请改用 _mpyu4ll
- _smpy2 不受支持，请改用 _smpy2ll

以下对齐的内存访问内在函数不会对齐输入地址：

- amem2 不会对齐地址
- amem2_const 不会对齐地址
- amem4 不会对齐地址
- amem4_const 不会对齐地址
- amem8 不会对齐地址
- amem8_const 不会对齐地址
- amemd8 不会对齐地址
- amemd8_const 不会对齐地址

2.9 伽罗瓦域乘法指令

伽罗瓦域乘法指令的接口在 C7000 上有所不同。因此，需要手动调整包含这些指令和内在函数的代码：

- GMPY / _gmpy()
- GMPY4 / _gmpy4()
- XORMPY / _xormpy()

2.10 有关迁移代码的性能注意事项

平均而言，从 C6000 移植到 C7000 的代码往往在 C7000 器件上逐周期地以大致相同或更快的速度运行。但是，对于给定的一段代码，性能可能会大幅提高或降低，具体取决于代码的性质。C7000 ISA 与 C6000 存在某些差异，这些差异可能会对性能产生积极或消极的影响。这些差异取决于代码的编写方式以及编译器如何优化和矢量化代码。

下面的几个子节介绍了其中一些差异以及如何处理它们。

2.10.1 UNROLL Pragma

使用 UNROLL pragma 为 C6000 优化的源代码可能不允许 C7000 编译器完全矢量化循环。

我们建议您删除 UNROLL pragma 或增加与 UNROLL pragma 一同使用的因子。根据代码的编写方式，删除 UNROLL pragma 可能允许编译器矢量化循环并利用 C7000 的全矢量宽度。

2.10.2 子矢量访问

在 C6000 器件上访问矢量类型的一部分可能是“无阻碍的”，但在 C7000 器件上则需要额外的指令。

例如，在 C6000 上访问 int4 元素的子矢量可能是无阻碍的，因为 C6000 上的一个 int4 由四个 32 位寄存器组成。因此，编译器可以通过使用适当的 32 位寄存器来访问 int4 的一个元素。但是，在 C7000 器件上，int4 元素位于单个矢量寄存器中。因此，访问 int4 的一个元素需要编译器使用指令（例如 VGETW）来提取该数据。

类似地，在 C6000 器件上打包一个 int4 矢量可能是无阻碍的或基本无阻碍，而在 C7000 器件上则可能需要一系列指令（例如 VPUTW）。

如果 C7000 编译器能够进一步矢量化代码，则在某些情况下可能会减轻性能损失。例如，通过 _loll() 访问 64 位的低 32 位可以矢量化成 VDEAL2W。

2.10.3 16x16 和 16x32 位乘法

C7000 不支持访问源字（MPYH、MPYHL 等）的高 16 位的 16 位 x 16 位乘法。模拟可能进行右移的此类访问运算。请注意，访问和乘法可以由编译器矢量化。

C7000 使用 C6MPYHIR 和 C6MPYLIR 指令执行 16 位 x 32 位乘法（MPYHI、MPYLI 等）。这些指令没有矢量形式，因此在循环中使用这些指令可能会限制或消除编译器的矢量化。

旧文本：C7000 不支持访问源字（MPYH、MPYHL 等）的高 16 位的 16 位 x 16 位乘法。在乘法之前使用右移来模拟此类运算。同样，不支持 16 位 x 32 位乘法（MPYHI、MPYLI 等）。在乘法运算之前使用符号扩展来模拟此类运算。

2.10.4 __x128_t 类型

在循环中使用 `__x128_t` 类型和变量可能会显著限制或消除编译器的矢量化。

2.10.5 无符号数组偏移

C7000 不支持在寄存器中使用 32 位无符号值作为加载或存储的偏移量。如果无符号整数用于数组偏移量，编译器可能需要这样做。如果枚举类型用于数组偏移并且枚举类型的所有成员都是正数，也可能发生这种情况。这样的加载/存储将使用单独的偏移调节、加法和加载/存储指令进行模拟。此类模拟可能会影响性能。

您可以对数组偏移使用 32 位有符号整数类型，以避免这种行为。

2.10.6 流引擎和流地址生成器

请尽可能使用流引擎和流地址生成器。使用这些功能将需要对您的代码进行一些修改。有关详细信息，请参阅 *C7000 优化编译器用户指南* (SPRUIG8) 中的“流引擎和流地址生成器”部分。

2.10.7 其他优化指南

其他优化指南可以在 *C7000 C/C++ 优化指南* (SPRUIV4) 中找到。

3 主机仿真

可以在主机系统上仿真已迁移到 C7000 的 C6000 源代码，无论是通过 `c6x_migration.h` 头文件还是直接使用 C7000 内在函数和定义。

主机仿真允许使用不同的调试和编程环境。更多详细信息，请参阅 *C7000 主机仿真用户指南* (SPRUIG6)。

4 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 JANUARY 21, 2022 to MARCH 31, 2023 (from Revision D (January 2022) to Revision E (March 2023)) **Page**

- 删除了对 SPRUIG7 的引用，SPRUIG7 已合并到 SPRUIG8 中。 2

Changes from August 31, 2019 to January 21, 2022 (from Revision C (August 2019) to Revision D (January 2022)) **Page**

- 更新了本文档中的表格、图和交叉参考的编号格式。 1
- 添加了使用优化级别 3 的建议 3
- 512 位矢量大小限制适用于 C7100 和 C7120 3
- 提供了有关控制寄存器符号列表位置的信息。 5
- 添加了对性能预期和改进的建议。 8
- 添加了有关迁移使用 UNROLL pragma 的代码的信息 8
- 添加了有关迁移访问矢量类型元素的代码的信息。 8
- 添加了有关迁移使用 __x128_t 类型的代码的信息。 8
- 添加了将流引擎和流地址生成器与迁移代码一同使用的建议。 9
- 添加了参阅 C7000 C/C++ 优化指南的建议。 9

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