User’s Guide

# C6000-to-C7000 Migration

TEXAS INSTRUMENTS

**ABSTRACT**

This document provides information to assist you in migrating existing C6000™ source code to code written for the C7000™ (C7x) processor family. This will allow it to be compiled by the C7000 compiler. This process requires no additional tools; it only requires attention to how compiler features differ between the C6000 and C7000 processor families.

## Table of Contents

## 1 About This Document

Changes described in this document should be made to source code originally written for the C6000 family in order to migrate to C7000 (C7x) processors. In particular, this document describes aspects of source code that you will need to evaluate and manually migrate.

This document also describes the support provided by the `c6x_migration.h` header file, which is included in C7000 Run-Time Support Library. You may include this file to facilitate compilation. For applications that do not have hardcoded references to addresses, control registers, or memory-mapped peripherals, #including this header file should be sufficient to build and run.

This document is not intended to be a compiler user's guide for either the C6000 or C7000 compiler toolchain. Familiarity with the C6000 compiler is therefore assumed.

There are two major C6000 programming paradigms that are not supported by the C7000 compiler and are therefore not discussed in this document:

- C6000 linear assembly
- C6000 hand-coded assembly

Existing C6000 source code written in either of these formats will need to be rewritten either in C or in C7000 assembly in order to be compiled by the C7000 compiler.

### 1.1 Related Documents

The following documents will provide related information for the C7x:

- *C7000 C/C++ Compiler Getting Started Guide* (SPRUIG7)
- *C7000 C/C++ Optimizing Compiler User's Guide* (SPRUIG8)
- *C7000 C/C++ Optimization Guide* (SPRUIV4)
- *C6000 C/C++ Optimizing Compiler User's Guide* (SPRUI04)
- *C7000 Embedded Application Binary Interface (EABI) Reference Guide* (SPRUIG4)
- *C7000 Host Emulation User's Guide* (SPRUIG6)
- *VCOP Kernel-C to C7000 Migration Tool User's Guide* (SPRUIG3)

### 1.2 Trademarks

C6000™ and C7000™ are trademarks of Texas Instruments.
OpenCL™ is a trademark of Apple Inc. used with permission by Khronos.
All trademarks are the property of their respective owners.

# 2 Migrating C Source from C6000 to C7000

The sections in this chapter describe changes that will need to be made to source code and support provided by the `c6x_migration.h` header file.

## 2.1 Compiler Options

Change the following compiler command-line options when porting C6000 code to C7000:

- Set the --silicon_version option to --silicon_version=7100. (Or replace the -mv6600, -mv6740, or -mv6400+ option with -mv7100.)
- The --interrupt_threshold (-mi) option will be ignored. C7000 C code is always interruptible.
- The --speculate_loads (-mh) option will be ignored. The C7000 compiler uses speculative load instructions for all loads except those to ioport variables/addresses.
- Specify the size/speed tradeoff option using the --opt_for_speed (-mf) option. Change the previously used --opt_for_space (-ms) option to the corresponding -mf option using Table 2-1. The --opt_for_space options do not exist for C7000. An --opt_for_speed option **must be used** instead, even if no --opt_for_space option was used for C6000.

**Table 2-1. Corresponding -ms and -mf Options**

| --opt_for_space (-ms) level (C6000 only) | --opt_for_speed (-mf) level (C6000 and C7000) | Description |
|---|---|---|
| No C6000 option | -mf5 | Maximum performance on C7000; Code size could be very large |
| no -ms option selected | -mf4 | Near maximum performance on C7000; Maximum performance on C6000 |
| -ms0 | -mf3 | Favor performance |
| -ms1 | -mf2 | Favor code size |
| -ms2 | -mf1 | Near minimum code size |
| -ms3 | -mf0 | Minimum code size |

Consider using the --opt_level=3 (-O3) compiler option. This option enables a high level of optimization by the compiler.

## 2.2 Native Vector Data Types

The C6000 compiler supports the use of native vector data types, which are documented in Section 7.4.2 of the *TMS320C6000 Optimizing Compiler Users Guide* (SPRUI04).

When porting source code that relies on native vector data types to the C7000 compiler, understand the following differences:

- There is no `c6x_vec.h` file. Replace #includes of `c6x_vec.h` and `c6x.h` with `c7x.h` alone
- The --vectypes compiler option is on by default for C7000.
- Use --vectypes=off if you have symbols that conflict with the names of the native vector types.

On C6000, OpenCL™-like native vector data types had to be enabled using the --vectypes option. However, with the C7000 compiler, all native vector data types are enabled by default. Therefore, there is no need to use --vectypes=on. If you have existing code with symbols that conflict with the native vector data type symbols, you can turn off compiler recognition of those symbols by using the --vectypes=off option.

Note that there are two names for each native vector data type: one without a double-underscore prefix (for example, `int4`) and one with a double-underscore prefix (for example, `__int4`). The double-underscore versions of the native vector data types are always recognized by the compiler. The --vectypes=off option turns off only those vector data types that do not have the double underscore prefix.

For the best possible future compatibility and portability, we recommend that you rename any existing typedefs, structs, or classes (that are not intended to be native vector data types) that use the names of OpenCL and OpenCL-like native vector data types. This also enables use of the shorter native vector type names, which do not use the double-underscore prefix.

Because the C7100 and C7120 have 512-bit vector sizes, the maximum number of elements in a vector is larger than for the C6000. The C6000 is limited by the 16-element limitation imposed by OpenCL. Vector lengths for the C7000 are limited to the maximum elements shown in Table 2-2.

**Table 2-2. C7000 Supported Vector Types**

| Type | Description | Maximum Elements |
|---|---|---|
| char*n* | A vector of *n* 8-bit signed integer values | 64 |
| uchar*n* | A vector of *n* 8-bit unsigned integer values | 64 |
| short*n* | A vector of *n* 16-bit signed integer values | 32 |
| ushort*n* | A vector of *n* 16-bit unsigned integer values | 32 |
| int*n* | A vector of *n* 32-bit signed integer values | 16 |
| uint*n* | A vector of *n* 32-bit unsigned integer values | 16 |
| long*n* | A vector of *n* 64-bit signed integer values | 8 |
| ulong*n* | A vector of *n* 64-bit unsigned integer values | 8 |
| float*n* | A vector of *n* 32-bit single-precision floating-point values | 16 |
| double*n* | A vector of *n* 64-bit double-precision floating-point values | 8 |
| cchar*n* | A vector of *n* pairs of 8-bit signed integer values | 32 |
| cshort*n* | A vector of *n* pairs of 16-bit signed integer values | 16 |
| cint*n* | A vector of *n* pairs of 32-bit signed integer values | 8 |
| clong*n* | A vector of *n* pairs of 64-bit signed integer values | 4 |
| cfloat*n* | A vector of *n* pairs of 32-bit floating-point values | 8 |
| cdouble*n* | A vector of *n* pairs of 64-bit floating-point values | 4 |

The C6000 64-bit longlong*n*, ulonglong*n*, and clonglong*n* vector types are not supported on C7000. As long as the `c6x_migration.h` file is included, the compiler will map these types to the corresponding types that are supported by C7000: long*n*, ulong*n*, and clong*n* respectively.

## 2.3 Type Qualifiers: near and far

The `near` and `far` keywords are unneeded and will be ignored by the C7000 compiler. We recommend that you remove all instances of these keywords.

## 2.4 64-bit long Type

The `long` type on C7000 is 64 bits and corresponds to the LP64 model.

The `long` type on C6000 is 32 bits.

We recommend that your code use the C standard integer types `int64_t` and `int32_t` (et al.) when specific data type sizes are needed for portability across different machines and compilers. These standard integer types are defined in `stdint.h`, which is included as part of the C standard library support included the Runtime Support Library.

(The `int` type is 32 bits on both C6000 and C7000.)

## 2.5 References to Control Registers

References to control registers in C and C++ will need to be changed manually. The C7000 has a completely different set of control registers. Please see the *C7000 CPU and Instruction Set Reference Guide* for details.

Control register symbols supported by the compiler tools are listed in `c6x.h` for the C6000 Compiler Tools and in `c7x_cr.h` and `c7x_ecr.h` for the C7000 Compiler Tools. Control registers are declared in these header files using the `__cregister` keyword.

Common examples of code that requires changes are:

- **References to the C6000 Control Status Register (CSR).** This includes references to the saturation (SAT) bit. For C7000, this is now a bit in the Flag Status Register (FSR). Note that the SAT bit interface is provided only to ensure compatibility for C6000-specific code. Referencing the SAT bit when writing new C7000 code is deprecated.

  The SAT bit can be accessed using the `__get_C7X_FSR()` API, which is defined in `c6x_migration.h`. An 8-bit value is returned in which the SAT bit is designated as "Bit 7".
- **References to the Floating-Point Configuration Registers (FADCR, FAUCR, FMCR).** Bits pertaining to floating-point operations are now bits in the C7000 Flag Status Register (FSR) and the Floating Point Control Register (FPCR).

  The Floating Point Status bits can be accessed using the `__get_C7X_FSR()` API, which is defined in `c6x_migration.h`. An 8-bit value is returned in which the floating point status bits comprise bits 0-6.

## 2.6 Memory-Mapped Peripherals

Any use of memory-mapped registers and peripherals must be investigated and possibly changed.

- Check the documentation for the appropriate SDK or CSL (Chip Support Library) to determine if a call to a peripheral needs modification.
- If the code uses hard-coded addresses for a peripheral interface, those addresses will likely need to be changed.
- If the code declares a memory-mapped pointer to a peripheral's control registers, must make sure that the `volatile` keyword is used when declaring/defining the pointer variable. This allows the compiler use the appropriate memory instructions to access memory-mapped data. (The compiler needs to know this information so that a regular, non-speculative load is used.)

We recommend that you use the appropriate SDK and CSL for your devices and consult the associated documentation.

## 2.7 Run-Time Support

When compiling code written for C6000 with the C7000 compiler, you must #include the C6000-to-C7000 migration reference header file `c6x_migration.h` at the beginning of the migrated source file.

- For most applications, including this header file should be sufficient to build and run.
- This file is provided as a reference implementation. You can modify or rename the file. For example, renaming the file to `c6x.h` would remove the need to change many #include directives in a project.
- You may #include both `c6x_migration.h` and `c7x.h` while in transition between C6000 and C7000 code.
- C/C++ source code that does not rely on C6000 intrinsics does not require a migration header file.

If you want to remove all references to C6000 when migrating, it is not necessary to include the C6000-to-C7000 migration reference header file. Instead, remove or modify all references to C6000-specific intrinsics and definitions. In this case, replace all instances of `#include <c6x.h>` with `#include <c7x.h>`.

## 2.8 Contents of Migration Header File c6x_migration.h

The following sections outline the important parts of the provided `c6x_migration.h` header file and how to use it effectively.

### 2.8.1 Supported Macros

The `c6x_migration.h` migration header file redefines macros that were defined internally by the C6000 compiler toolchain. The definitions map to the appropriate C7000 definitions.

- **C6000 Target Macros:** All of the following target macros map to __C7000__.
  - __TMS320C6X__
  - _TMS320C6X
- **C6000 Subtarget Macros:** All of the following subtarget macros currently map to __C7100__.
  - _TMS320C6600
  - _TMS320C6740
  - _TMS320C6700_PLUS
  - _TMS320C67_PLUS
  - _TMS320C6700
  - _TMS320C64_PLUS
  - _TMS320C6400_PLUS
  - _TMS320C6400
- **Endian Macros:** The following are deprecated. The C7000 compiler defines __big_endian__ or __little_endian__, which should be used instead.
  - _BIG_ENDIAN
  - _LITTLE_ENDIAN
- **EABI Macros:** The following is deprecated. __TI_EABI__ should be used instead.
  - __TI_ELFABI__

### 2.8.2 Non-Supported Macros

The following macros will not be defined by the `c6x_migration.h` migration header file. You should either change the code or manually defining the macro via the --define (-D) compiler option.

- __DSBT__ — No support.
- __TI_TLS__ and __TI_USE_TLS — Not yet implemented.
- __TI_32_BIT_LONG__ and __TI_40_BIT_LONG__ — Neither of these macros are defined since `long` on C7000 is always 64 bits.
- _LARGE_MODEL, _SMALL_MODEL, _LARGE_MODEL_OPTION — These C6000 macros rely upon various memory model options which are no longer supported on C7000. If your code requires that these macros be defined, use the --define (-D) compiler option.

### 2.8.3 Legacy Data Types

Some source files may reference the following data types.

- `__float2_t` — This is a "container" for 2 float values. It is typedefed to `double` in both C6000 and C7000. All C6000 intrinsics that work with __float2_t are declared in `c6x_migration.h`.
- `__x128_t` — On C6000, this is a vector “container" type, a special 128-bit sized struct vector. On C7000, this type is defined in `c6x_migration.h`. All C6000 intrinsics that work with __x128_t are declared in `c6x_migration.h`.
- `__int40_t` — On C6000, this is a special first-class integer type, like `int` and `short`. It has 40 bits of precision. On C6000, it is valid to use this type in native operations (such as +, -) as well as with intrinsics. This type is not defined for C7000, and so its corresponding operations are not supported. An intrinsic for 40-bit saturation of a 64-bit value is available (VSATLW) for C7000.

### 2.8.4 Legacy Intrinsics

The C6000 intrinsic names, which are defined in c6x.h, do not conflict with any C7x intrinsic names. Therefore, including both the migration header `c6x_migration.h` as well as `c7x.h` will not cause an issue. Each C6000 intrinsic is mapped to either a single C7000 instruction or a set of C7000 instructions that perform or emulate the same behavior.

- If a C6000 intrinsic is mapped to a single C7000 instruction, then search `c7x.h` for the C7000 C-idiom for that instruction.
- If a C6000 intrinsic is mapped to a set of instructions, then an example C7000 C-idiom will be providedin `c7x.h`.

For example, in `c6x_migration.h`, the `_dadd` intrinsic is declared and the mapped C7000 instruction, `VADDW` , is indicated in the comment above the declaration:

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

In `c7x.h`, the same instruction is shown as well as its supported C-idiom, whether that is a C intrinsic or operator:

```
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;
```

As another example, the `_unpkbu4` intrinsic is declared, but there is no single C7000 instruction to which it corresponds. So, `c7x.h` shows the C7000 C-idiom as follows:

```
/*-----------------------------------------------------------------------------*/
/* _unpkbu4 uses the VUNPKLUB and VUNPKHUB to unpack the low and high 2      */
/* bytes of the argument, and then constructs the result. An equivalent C7X  */
/* piece of code would look like:                                             */
/*                                                                            */
/* ushort4 _unpkbu4(uchar4 src)                                               */
/* {                                                                          */
/*   ushort4 dst;                                                             */
/*   dst.lo = __unpack_low(src);                                              */
/*   dst.hi = __unpack_high(src);                                             */
/*   return dst;                                                              */
/* }                                                                          */
/*-----------------------------------------------------------------------------*/
long long __BUILTIN _unpkbu4(unsigned)
```

The following deprecated C6000 intrinsics are not supported with the C7000 Compiler Tools. Use the `long long` variants instead:

- _mpy2 is not supported, use _mpy2ll instead
- _mpyhi is not supported, use _mpyhill instead
- _mpyli is not supported, use _mpylill instead
- _mpyid is not supported, use _mpyidll instead
- _mpysu4 is not supported, use _mpysu4ll instead
- _mpyu4 is not supported, use _mpyu4ll instead
- _smpy2 is not supported, use _smpy2ll instead

The following aligned memory access intrinsics will not align input addresses:

- amem2 will not align address
- amem2_const will not align address
- amem4 will not align address
- amem4_const will not align address
- amem8 will not align address
- amem8_const will not align address
- amemd8 will not align address
- amemd8_const will not align address

## 2.9 Galois Field Multiply Instructions

The interface to Galois Field Multiply Instructions is different on C7000. Therefore, code containing these instructions and intrinsics will need to be adjusted manually:

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

## 2.10 Performance Considerations for Migrated Code

On average, code ported from C6000 to C7000 tends to run at about the same speed or better on C7000 devices, on a cycle-for-cycle basis. However, for a given piece of code, there can be large performance increases and decreases, depending on the nature of the code. The C7000 ISA has certain differences from C6000 that can positively and negatively affect performance. These differences depend on how the code is written and how that code can be optimized and vectorized by the compiler.

The subsections that follow explain some of those differences and what to do about them.

### 2.10.1 UNROLL Pragma

Source code optimized for the C6000 using the UNROLL pragma may not allow the C7000 compiler to fully vectorize a loop.

We recommend that you either remove the UNROLL pragma or increase the factor used with the UNROLL pragma. Depending on how the code is written, removing the UNROLL pragma may allow the compiler to vectorize the loop and utilize the full vector width of the C7000.

### 2.10.2 Subvector Access

Accessing a portion of a vector type may be “free” on C6000 devices, but requires an extra instruction on C7000 devices.

For example, a subvector access of an int4 element is likely to be free on C6000, since an int4 on C6000 is composed of four 32-bit registers. Therefore, accessing one element of an int4 can be performed by the compiler by using the appropriate 32-bit register. However, on C7000 devices, an int4 element is located in a single vector register. Therefore, accessing one element of an int4 requires the compiler to use an instruction (such as VGETW) to extract that data.

Similarly, packing a vector of int4 is likely free or almost free on C6000 devices, while on C7000 devices it may require a sequence of instructions (such as VPUTWs).

If the C7000 compiler is able to vectorize the code further, in some cases the performance penalty may be mitigated. For example, the access of the low 32 bits of 64 bits with _loll() may be vectorized into VDEAL2W.

### 2.10.3 16x16 and 16x32 Bit Multiplies

C7000 does not support 16-bit x 16-bit multiplication that accesses the upper 16 bits of a source word (MPYH, MPYHL, etc.). Such access operations are emulated, which may use a right shift. Note that the access and the multiply may be vectorized by the compiler.

C7000 performs 16-bit x 32-bit multiplication (MPYHI, MPYLI, etc.) using the C6MPYHIR and C6MPYLIR instructions. There are no vector forms of these instructions, so using these instructions in a loop may limit or eliminate vectorization by the compiler.

Old text: C7000 does not support 16-bit x 16-bit multiplication that accesses the upper 16 bits of a source word (MPYH, MPYHL, etc.). Such operations are emulated using a right shift before the multiply. Similarly, 16-bit x 32-bit multiplies are not supported (MPYHI, MPYLI, etc.). Such operations are emulated with sign extensions before the multiply operations.

#### 2.10.4 __x128_t Type

Use of `__x128_t` types and variables in a loop may significantly limit or eliminate vectorization by the compiler.

#### 2.10.5 Unsigned Array Offsets

C7000 does not support using a 32-bit unsigned value in a register for an offset to a load or store. The compiler may need to do this if an unsigned integer is used for an array offset. This may also occur if an enum type is used for an array offset and all members of the enum type are positive. Such loads/stores will be emulated using individual offset scaling, addition, and load/store instructions. This emulation may affect performance.

You can avoid this behavior by using the 32-bit signed integer type for array offsets.

#### 2.10.6 Streaming Engine and Streaming Address Generator

Consider using the Streaming Engine and Streaming Address Generator when possible. Using these features will require some rework of your code. For more information, see the “Streaming Engine and Streaming Address Generator” section in the *C7000 Optimizing Compiler User’s Guide* (SPRUIG8).

#### 2.10.7 Additional Optimization Guidance

Additional optimization guidance can be found in the *C7000 C/C++ Optimization Guide* (SPRUIV4).

## 3 Host Emulation

It is possible to emulate C6000 source code that has been migrated to C7000 on a host system, whether by way of the `c6x_migration.h` header file or directly using C7000 intrinsics and definitions.

Host emulation allows for the use of different debug and programming environments. Please refer to the *C7000 Host Emulation User Guide* (SPRUIG6) for details.

## 4 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

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

- Updated the numbering format for tables, figures, and cross-references throughout this document................. 1
- Added recommendation to use optimization level 3........................................................................... 3
- The 512-bit vector size limit applies to the C7100 and C7120.......................................................... 3
- Provided information about location of lists of control register symbols............................................ 5
- Added recommendations for performance expectations and improvement...................................... 8
- Added information about migrating code that uses the UNROLL pragma.........................................8
- Added information about migrating code that accesses elements of a vector type..........................8
- Added information about migrating code that uses the __x128_t type..............................................9
- Added recommendation to use Streaming Engine and Streaming Address Generator with migrated code...... 9
- Added recommendation to refer to the C7000 C/C++ Optimization Guide........................................9