*User's Guide*

# *C7000 Host Emulation v5.0.0.LTS User's Guide*


**ABSTRACT**

C7000™ Host Emulation lets you use C7000 compiler intrinsics and TI vector types on a PC or Linux® host system. This allows you to use different debugging tools and programming environments to prototype programs targeted for C7000 hardware before using the C7000 compiler. The Host Emulation package does not attempt to simulate the C7000 CPU.


## Table of Contents

# 1 About This Document

This document serves as a user's guide for writing C7000 DSP programs using C7000 Host Emulation. Included are examples that outline the key differences between programming with the C7000 compiler (cl7x) and programming using the Host Emulation package on a desired host system. The purpose of this document is to provide a reference of the key features and limitations of the C7000 Host Emulation package.

## 1.1 Related Documentation

The following documents provide related information for C7000:

- *C7000 C/C++ Optimizing Compiler User's Guide* (SPRUIG8)
- *C7000 C/C++ Optimization Guide* (SPRUIV4)
- *C7000 Embedded Application Binary Interface (EABI) Reference Guide* (SPRUIG4)
- *C6000-to-C7000 Migration User's Guide* (SPRUIG5)
- *VCOP Kernel-C to C7000 Migration Tool User's Guide* (SPRUIG3)
- *C7x Instruction Guide* (SPRUIU4, which is available through your TI Field Application Engineer)
- *C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator* (SPRUIP0, which is available through your TI Field Application Engineer)
- *C71x DSP Corepac Technical Reference Manual* (SPRUIQ3, which is available through your TI Field Application Engineer)

## 1.2 Disclaimer

The C7000 Host Emulation support is an experimental product. It is recommended that users read and understand all of the limitations disclosed in this document. Additional limitations may exist that are not disclosed in this document.

## 1.3 Trademarks

# 2 Getting Started with Host Emulation

The C7000 Host Emulation package consists of C++ source and header files used to drive the features provided by the C7000 compiler.

Depending on the desired host, the source files may need to be built on the host prior to compiling a C7000 program. Detailed instructions on how to build source on different hosts are provided in the sections that follow.

Familiarity with the *C7000 C/C++ Optimizing Compiler User's Guide* (SPRUIG8) and the C7000 Runtime Support Library is required to fully understand the content in this guide and to use Host Emulation successfully.

## 2.1 System Requirements

In general, system requirements for C7000 Host Emulation match the system requirements needed to install the C7000 Code Generation Tools (CGT).

The pre-compiled libraries that are shipped with the C7000 Host Emulation package require the following compiler installations. It is recommended that you use a compiler version that matches the version that was used to build and test C7000 Host Emulation.

- **Linux** (x86-64 bit)
  - GNU g++ compiler. C7000 Host Emulation was built with version 9.5.0.
- **Microsoft Windows®** (x86-64 bit)
  - Visual C++ build tools (standalone or packaged with corresponding Visual Studio® IDE installation). C7000 Host Emulation was built with C++ toolset v143, compiler version 14.37.
  - GNU g++ compiler. C7000 Host Emulation was built with version 9.2.0 from MSYS2.

Boost C++ libraries and headers are not required in order to use host emulation.

## 2.2 Installation Instructions

The C7000 Host Emulation package will be distributed as a part of the C7000 CGT. Installing C7000 CGT on a desired platform will install the C7000 Host Emulation package as well.

Libraries for different platforms and compilers can be found in the `host_emulation` directory of the installed tools. All header files associated with Host Emulation can be found in the `host_emulation/include` directory of the installed tools.

For Visual C++, the *<target>*`-host-emulation.lib` library is compatible with the *release* version of the static run-time library. The *<target>*`-host-emulationd.lib` library is compatible with the *debug* version of the static run-time library.

## 2.3 Host Emulation Library Versions

Multiple host emulation libraries and include directories are provided in the `host_emulation` directory of the installation. Each of these libraries corresponds to a different combination of --silicon_version and --mma_version command line options that can be used with the C7000 compiler. When using C7000 Host Emulation, build with the library and include directory that correspond to the desired compiler options as follows:

**Table 2-1. Host Emulation Builds**

| --silicon_version option | --mma_version option | Host Emulation Library |
|---|---|---|
| 7100 | 1 | C7100 |
| 7120 | 2 | C7120 |
| 7504 | 2_256 | C7504 |
| 7524 | 2_256 | C7524-MMA2_256 |
| 7524 | 2_256f | C7524-MMA2_256F |

## 2.4 Summary of Differences: Host Emulation Coding vs. Native C7000 Coding

When coding an application to run with C7000 Host Emulation, you should be aware of the following general limitations:

- All source files must #include the `c7x.h` file. (See Section 3.1.)
- Use of standard integer types rather than built-in types is recommended for future portability. (See Section 3.2.)
- The code must use C++14 due to the underlying implementation, which relies heavily on C++14 constructs and features. (See Section 3.2.)
- C7000 pragmas are not supported with Host Emulation. (See Section 3.2.)
- There are certain limitations and differences with intrinsics. (See Section 4.) For example, intrinsics that operate directly on memory and the L1D cache cannot be used with C7000 Host Emulation. (See Section 4.4.)

See Section 10 for information about specific compiler errors and warnings and about syntax interpretation differences between the C7000 compiler and the Host Emulation compiler.

# 3 General Coding Requirements

## 3.1 Required Header Files

Regardless of your chosen host, certain prerequisites are required for every program written to be run with C7000 Host Emulation.

All source files that use C7000 compiler features with Host Emulation need to #include the `c7x.h` or `c6x_migration.h` file, as appropriate. These files in turn include all other required header files. When compiling for Host Emulation, do not #include any of the other header files provided in the C7000 Run Time Support Library.

When compiling for Host Emulation, do not #include any of the headers found in the C7000 Run Time Support library. This includes the `c7x.h` and `c6x_migration.h` files. Instead, use preprocessor symbols to control which header files are included.

**Table 3-1. Host Emulation Header Files**

| File Included Explicitly | Description |
|---|---|
| c7x.h | Main header file. Includes all others listed below except c6x_migration.h. |
| c6x_migration.h | Legacy intrinsics and data types. Includes all others listed below. |
| **Files Included Automatically** | |
| c7x_cr.h | Global control register definitions |
| c7x_ecr.h | Global extended register definitions |
| c7x_iluthist.h | Internal lookup table and histogram control interface |
| c7x_luthist.h | Lookup table and histogram control interface |
| c7x_strm.h | Streaming engine control interface |

The `ti_he_impl` folder contains other header files used for the implementation; these files should not be included directly.

## 3.2 Package Dependencies

Programs written for C7000 Host Emulation must use the C++14 language due to the underlying implementation, which relies heavily on C++14 constructs and features.

Depending on the compiler, a special flag to enable C++14 support may be required in the compilation command.

While not mandated, it is highly encouraged that you use standard integer types (such as `int32_t` ) when programming using C7000 Host Emulation. Usage of built-in data types may compile and run, but these results cannot be guaranteed to be correct on all platforms. Using standard integer types in place of the corresponding built-in type will achieve correct results and will have no effect on the ability to transition the program to the C7000 compiler.

Use of C7000 compiler attributes and directives will create undefined warnings when using Host Emulation. This behavior is expected and cannot be remedied. If these attributes and directives are required for the program to run on a target chip, the warnings can typically be suppressed on the Host Emulation compiler.

The C7000 Host Emulation package does not emulate C7000 compiler pragmas. As a result, C7000 compiler pragmas will have no effect when used in code run with C7000 Host Emulation.

A list of C7000 compiler symbols that are defined automatically when using Host Emulation is provided in Table 3-2.

**Table 3-2. C7000 Preprocessor Symbols**

| Defined Preprocessor Symbols | Description |
|---|---|
| __C7000__ | Defined if compiled for the C7000 target or any type of C7000 Host Emulation. |
| __C7100__ | Defined if compiled for C7100 Host Emulation. |
| __C7120__ | Defined if compiled for C7120 Host Emulation. |
| __C7504__ | Defined if compiled for C7504 Host Emulation. |
| __C7524__ | Defined if compiled for C7524 Host Emulation. |
| __C7X_HOSTEM__ | Defined if compiled for Host Emulation. This is not defined when the target compiler (cl7x) is used. |
| __C7X_MMA__ | Defined by default. |
| __C7X_MMA_1__ | Defined if compiled for Host Emulation with MMA 1 support. |
| __C7X_MMA_2__ | Defined if compiled for Host Emulation with MMA 2 support. |
| __C7X_MMA_2_256__ | Defined if compiled for Host Emulation with MMA 2_256 support. |
| __C7X_MMA_2_256F__ | Defined if compiled for Host Emulation with MMA 2_256F support. |
| __C7X_NUM_SE__ | Defined to the number of Streaming Engines that are available. Currently always 2. |
| __C7X_NUM_SA__ | Defined to the number of Streaming Address Generators that are available. Currently always 4. |
| __little_endian__ | Defined by default. |

## 3.3 Example Program

The following is a sample program that can be compiled using both Host Emulation and the C7000 compiler interchangeably without modification to the source. A sample compiler command is provided for each case.

The C7000 compiler (cl7x) command-line options are not compatible with the Host Emulation compilers.

```
/* 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 Host Emulation compiler command (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
```

The -fno-strict-aliasing command-line option should always be used with g++ when using Host Emulation. This option ensures the g++ compiler does not use type differences to make aliasing decisions. The Host Emulation implementation uses differing types in order to implement TI vector types. Therefore if this option isn't used, g++ may incorrectly optimize TI vector code utilizing the Host Emulation feature, which may lead to unexpected and incorrect results.

### C7000 compiler command:

```
cl7x test.cpp
```

# 4 Intrinsics

All intrinsics that are available with the C7000 compiler are available for use with C7000 Host Emulation. The following subsections address possible issues when using the following types of intrinsics with Host Emulation:

- OpenCL-Like intrinsics (see Section 4.1)
- Intrinsics used to program the streaming engine and streaming address generator (see Section 4.2)
- Intrinsics used to migrate legacy code written for the C6000™ compiler (cl6x) (see Section 4.3)
- Intrinsics that act on the memory system (see Section 4.4 for differences)

The following additional types of intrinsics are the same for C7000 Host Emulation and the C7000 compiler: intrinsics used for special loading and storing of vector and scalar elements, low-level direct-mapped intrinsics, intrinsics that are a part of the vector predicate to register interface, and intrinsics used to perform lookup table and histogram operations.

Intrinsics that modify control registers will do so in C7000 Host Emulation. All control registers that are available under C7000 Host Emulation can be referenced at any time as an unsigned 64-bit integer.

Reading and writing registers that rely on hardware information, such as execution mode and cycle count, is not fully supported in Host Emulation. While all symbols and intrinsics associated with these registers are defined for compilation purposes, their values cannot be depended upon to be accurate when using Host Emulation.

Some intrinsics may require special handling to be used properly. For all intrinsics not mentioned in the subsections that follow, their functionality remains exactly as it is on C7000. A comprehensive list of the intrinsics available for use with the C7000 compiler can be found in the `c7x.h` file and the other header files provided in the C7000 Runtime Support Package.

Instruction execution emulates the hardware as closely as possible.

## 4.1 OpenCL-Like Intrinsics

All OpenCL™-like intrinsics available in the C7000 compiler are available for use in C7000 Host Emulation. The intrinsic interface remains unchanged; any legal use of an OpenCL-like intrinsic is also legal in C7000 Host Emulation.

## 4.2 Streaming Address Generator Intrinsics

All streaming address generator intrinsics that are available with the C7000 compiler are also available for use in C7000 Host Emulation. Their interface is the same as it is with the C7000 compiler.

## 4.3 C6000 Legacy Intrinsics

All legacy intrinsics defined in `c6x_migration.h` are available for use in C7000 Host Emulation. Their interface is the same as it is with the C7000 compiler.

Section 8 discusses requirements regarding legacy data types and assumptions about their SIMD usage. As a result of those limitations, all legacy data types must be treated as container types. That is, all initialization and interaction with legacy data types must be through intrinsics. Section 8 also contains examples of how to program with legacy data types and intrinsics when using C7000 Host Emulation. The *C6000-to-C7000 Migration User's Guide* (SPRUIG5) and the `c6x_migration.h` header file should be used as references any time C6000 code is used within a C7000 program.

## 4.4 Memory System Intrinsics

The intrinsics listed in Table 4-1 have no effect when used with Host Emulation. These intrinsics operate on memory and the L1D cache, which cannot be emulated on a host system.

**Table 4-1. Memory System Intrinsics**

| Intrinsic Name | Implementation Note |
|---|---|
| __memory_fence | Executes successfully with no effect |
| __memory_fence_store | Executes successfully with no effect |
| __prefetch | Executes successfully with no effect |

# 5 TI Vector Types

The C7000 Host Emulation package generally allows for the use of TI vector types (for example, `int16`) to be used in the same way as with the C7000 compiler. Boolean vectors, such as bool16, are also supported.

However, due to C7000 Host Emulation being written in C++, there are limitations. The following sections discuss and provide examples of these limitations. Where limitations exist, usage and syntax changes may be required.

**Note**

If a TI vector type feature is not mentioned here but is permissible with the C7000 compiler, the feature *is permissible* with C7000 Host Emulation.

As with the C7000 compiler, C7000 Host Emulation enables support for vector data types by default. Instead of using the --vectypes=off C7000 compiler option to disable vector data type names, to disable vector data type names that are *not* prefixed with a double-underscore you should define the TI_VECTYPES_OFF macro. For example, the `__int4` type is always available, but defining TI_VECTYPES_OFF on the command line, such as with `g++ -DTI_VECTYPES_OFF`, disables the `int4` type. (The syntax to define a macro varies depending on your compiler.)

## 5.1 Constructors

As of the v3.0 of the C7000 compiler, the C7000 TI Compiler (cl7x) accepts constructor-style syntax for vector initialization. To create portable code that compiles using both cl7x and Host Emulation, use the constructor-style syntax for vector initialization instead of the "cast/scalar-widening" style of initialization, which works correctly only with cl7x.

Examples of the correct and incorrect vector constructor initialization syntax are shown in the following code.

```
/* 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 Accessors

C7000 Host Emulation provides the following supports for accessors:

- Single element accessors, such as .s0(), are supported.
- Half vector accessors, such as .lo() and .even(), are supported.
- Complex accessors, such as .r(), are supported.
- Subscript accessors, such as .s[0], are supported.
- Multiple element swizzle accessors, such as .s0312(), *are not supported*. This is because there are too many combinations of the swizzle accessors and it would not be possible to have definitions for all of them. Workarounds involve using one the following idioms.

```
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 Vector Operators

All vector operators are supported in Host Emulation, except the vector ternary operator—that is, when the condition expression is a vector.

All other operator implementations follow the specification detailed in the *C7000 C/C++ Optimizing Compiler User's Guide* (SPRUIG8). Illegal uses of an operator result in compiler errors. However, the type of message received may vary. In a few cases, illegal uses of some operators result in assertion errors at compile time rather than traditional compiler errors.

Nested subvector accesses should be specified using function-call syntax. For example, when compiling for Host Emulation, `vect.lo().lo()` is legal, but `vect.lo.lo` is not. As of v3.0 of the C7000 compiler, there is no limit to the nesting depth for subvectors compiled for Host Emulation.

## 5.4 Print Debug Function

A print function is provided with C7000 Host Emulation that can be used on any TI vector type. This function prints out a formatted list of the contents of the vector. This function is specific to C7000 Host Emulation and is not supported by the C7000 compiler. As a result, references to this function must be omitted or protected by checks of the __C7X_HOSTEM__ preprocessor symbol in order to be compiled using the C7000 compiler. The following example shows how the print function can be used at different accessor levels of a vector.

```
/* 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 Streaming Engine and Streaming Address Generator

The C7000 Host Emulation Streaming Engine (SE) and Streaming Address Generator (SA) interface is the same as with the C7000 compiler.

# 7 Lookup Table and Histogram Interface

The C7000 Host Emulation Lookup Table (LUT) and Histogram (HIST) interface is the same as with the C7000 compiler. Any intrinsic or definition mentioned in `c7x_luthist.h` is also defined and implemented in C7000 Host Emulation and can be used in the same way.

## 7.1 Lookup Table and Histogram Data

When using C7000 Host Emulation, a 32K portion of memory is allocated to represent the C7000's L1D cache for use with LUT and HIST operations. The symbol, `lut_sram` , should not be used directly under normal circumstances. Accessing `lut_sram` directly is analogous to accessing the C7000's L1D cache directly, which is prohibited. However, the symbol is available for debugging purposes.

# 8 C6000 Migration

All intrinsics and data types defined in `c6x_migration.h` are available in C7000 Host Emulation for migrating legacy code. All intrinsics that map to a C7000 instruction or a set of instructions are used in the same way as they are with the C7000 compiler. However, as mentioned in Section 4.4, there are limitations when using legacy types in C7000 Host Emulation.

The following sections focus only on the differences between using legacy code with the C7000 compiler and using C7000 Host Emulation. The *C6000-to-C7000 Migration User's Guide* (SPRUIG5) contains detailed information on migrating C6000 programs to C7000.

## 8.1 __float2_t Legacy Data Type

With the C7000 compiler, the `__float2_t` legacy type is treated as a `double` at all times. This is valid with the C7000 compiler as a double is 64-bits wide and can fit two 32-bit floating point elements for use with SIMD operations.

This is not the case when using host systems that execute on Intel x86 architectures. When performing loads and stores of doubles on Intel x86 machines, there is an automatic conversion that takes place to convert a 64-bit double to an 80-bit “extended-real” type. This presents a problem when a double is used to store two distinct 32-bit floating point values as normalization can occur on the 80-bit “extended-real” types, which changes the bits stored in memory. If an extension to an 80-bit type with normalization is done on a double that represents two 32-bit floating point types, then the data can no longer be guaranteed and SIMD operations that expect two floating point values will have inconsistent results.

To solve this problem, C7000 Host Emulation contains a separate class definition for the `__float2_t` type that is treated as an opaque container type. Container types can only be modified, accessed, and initialized using special intrinsics. While the `__float2_t` class definition contains public accessor methods, it is recommended that only intrinsics are used to modify `__float2_t` types as any member of the C7000 Host Emulation `__float2_t` type will be undefined with the C7000 compiler. The `__float2_t` class type should be used when a single data structure that represents two 32-bit floating point values is required in a legacy intrinsic. When writing C7000 Host Emulation code that utilizes C6000 legacy constructs, a `double` type should only be used to represent one double precision floating point value.

As a result of having a separate definition for the `__float2_t` type, the `_ftof2` intrinsic must be used to construct a `__float2_t` type. With the C7000 compiler, this intrinsic is defined as `_ftod` which creates a double type from two floating pointer arguments. The accessor methods for `__float2_t` are defined in the same manner.

Table 8-1 lists the intrinsics that are distinctly defined for C7000 Host Emulation. Despite the distinctions made in the definitions of the intrinsics listed in this table, legacy code written for C7000 Host Emulation can be transferred to the C7000 compiler without change.

**Table 8-1. Legacy Intrinsics with Distinct Definitions in Host Emulation**

| Intrinsic Name | Previous Definition | Function |
|---|---|---|
| _ftof2 | _ftod | Construct __float2_t type from 2 floating point values |
| _lltof2 | _lltod | Convert long long values to __float2_t type |
| _f2toll | _dtoll | Convert __float2_t type to long long |
| _hif2 | _hif | Access high 32-bit float from __float2_t type |
| _lof2 | _lof | Access low 32-bit float from __float2_t type |
| _fdmv_f2 | _fdmv | Alternative to using PACK instruction to construct __float2_type from 2 floats |
| _fdmvd_f2 | _fdmvd | Alternative to using PACKWDLY4 instruction to construct __float2_type from 2 flaots |
| _hif2_128 | _hid128 | Access high __float2_t type from __x128_t type |
| _lof2_128 | _lod128 | Access low __float2_t type from __x128_t type |
| _f2to128 | _dto128 | Construct __x128_t type from 2 __float2_t types |

The following examples construct and set `__float2_t` variables in valid and invalid ways as indicated in the comments.

```
/* __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 Matrix Multiply Accelerator (MMA) Interface

The C7000 Host Emulation Matrix Multiply Accelerator (MMA) interface is the same as the interface used with the C7000 compiler on the target hardware with one important difference. All intrinsics and definitions mentioned in `c7x_mma.h` are also defined and implemented for C7000 Host Emulation and can be used in the same ways. However, programs must explicitly indicate when the MMA state advances by calling the provided __HWAADV() intrinsic. This is because, unlike the target hardware, the MMA that is emulated for the host can't be tied to the notion of a CPU clock.

Programs must keep track of instructions that are intended to execute in parallel and explicitly advance the MMA state by calling __HWAADV() after each set of "parallel" instructions.

To make portability easier between host and target modes, the __HWAADV() intrinsic is defined as an empty macro by the target compiler.

# 10 Compiler Errors and Warnings

When using C7000 Host Emulation to program for C7000, compiler errors and warnings will differ from those seen when compiling the same code with the C7000 compiler. Due to the complex implementation of some of the C7000's features in Host Emulation, the following sections define some key terms needed to help decipher some Host Emulation compiler errors you may see.

This section also discusses Host Emulation compiler errors and warnings that may be emitted when attempting to use C7000 Host Emulation specific syntax and constructs. Cases that may not trigger compiler errors or warnings are also described.

## 10.1 Key Terms Found in Compiler Errors and Warnings

When dealing with TI vector constructors, compiler errors and warnings may reference different classes and their respective members. Table 10-1 lists these key terms and their purposes.

The only real difference for those types of errors is that instead of containing something like "_c70_he_detail::vtype<int, 16ul>", it will contain something like "_c70_he_detail::vtype<int, 16ul, **(_c70_he_detail::VTYPE_KIND)0**>".

**Table 10-1. Key terms found in vector-related compiler errors and warnings**

| Term | Purpose | Sample Error/Warning |
|---|---|---|
| _c70_he_detail | Namespace containing all vector classes and operators | "Error: could not convert '_c70_he_detail::vtype<long int, 8ul, (_c70_he_detail::vtype_kind)0>(0)'…" |
| Vtype | High-level vector class name | "Error: conversion from 'int2 {aka _c70_he_detail::vtype<int, 2ul, (_c70_he_detail::vtype_kind)0>}'…" |

## 10.2 Host Emulation Specific Syntax

C7000 Host Emulation both introduces and omits some syntax used with the C7000 compiler. While these differences are detailed throughout this document, the Host Emulation compiler cannot be relied on to emit warnings and errors in all of these cases. This is due to the fact that some of the original syntax allowed by the C7000 compiler constitutes legal C++ code, which the Host Emulation compiler would have no reason to warn the user about. While using the original C7000 compiler syntax in some cases may be syntactically correct, the results cannot always be guaranteed. Table 10-2 lists the host compiler errors and warnings, or lack thereof, which may arise when using the original C7000 syntax with C7000 Host Emulation.

**Table 10-2. Syntax change related compile errors and warnings**

| Description | Example | Compiler Output |
|---|---|---|
| Using cl7x cast-style constructor syntax with Host Emulation | `// Works with cl7x but not Host Emulation`<br>`(long8)(1,2,3,4,5,6,7,8)`<br>vs.<br>`// Recommended; works on Host Emulation and cl7x`<br>`long8(1,2,3,4,5,6,7,8)` | No errors or warnings. Results are incorrect/ unexpected using cast-style constructor syntax. |
| Ternary operator with vector as "boolean expression" | `res = vec1 ? vec2 : vec3` | Compiler error: "Cannot convert vec_type to bool". |
| Using swizzle accessor with member data syntax | `example.s0121` | Compiler error: "Member does not exist". |
| Using swizzle accessor with function data syntax | `example.a0121()` | Compiler error: "Member function does not exist" |
| Not using function syntax with accessors | `. . . = vect.s0;` | Compile time error. |
| Using an invalid value within SE/SA parameters | Setting VECLEN to a negative number | At run-time, Host Emulation will state which field is invalid. |

# 11 Revision History

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

**Changes from March 15, 2024 to March 15, 2025 (from Revision K (March 2024) to Revision L (March 2025))** **Page**

- Documented mapping from compiler command line options to Host Emulation libraries used to build for various device and MMA variants....3
- Added description of c7x_iluthist.h internal header file....5
- Added preprocessor symbols for C7524, MMA support, Streaming Engines, and Streaming Address Generators....5
- Vector data types may be disabled by defining TI_VECTYPES_OFF....8

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

- Updated compiler versions with which C7000 Host Emulation was built....3

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

- Corrected example for multiple element swizzle accessor workaround....8
- Corrected example code....9

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

- Added __C7X_HOSTEM__, __C7120__, and __C7504__ preprocessor symbols....5
- Simplified example program syntax since the cl7x compiler now accepts the constructor-style syntax for vector initialization....6
- Native vector types are now called "TI vector types." In addition, several limitations related to complex vector types have been resolved. See the *C7000 C/C++ Optimizing Compiler User's Guide* for details....8
- Both cl7x and Host Emulation compilers now accept constructor-style syntax for vector initialization....8
- Clarified types of accessors supported, and updated workarounds for swizzle accessors....8
- Modified example code....9
- Updated table of key terms found in vector-related errors and warnings....15
- Updated table of errors and warnings related to syntax changes....15

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

- Added a list of pointer comparisons operators to vector and complex element pointer types....8

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

- Allow Visual C++ build tools versions after 2017....3
- Both release and debug-compatible static run-time libraries are provided for use with Microsoft Visual C++....3

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



**Table 12-1. Changes from January 28, 2020 to May 1, 2020 (from Revision D to Revision E)**

| Version Added | Location | Notes |
|---|---|---|
| SPRUIG6E | Section 1.1 | Added *C7x Instruction Guide* and other documents to list of related documents. |
| SPRUIG6E | -- | Added information and workaround for incompatible argument types for load and store intrinsics. |
| SPRUIG6E | Section 5 | Additional memory is required for vector data types with Host Emulation. |
| SPRUIG6E | Section 5.3 | Nested subvectors are limited to a depth of 2 with Host Emulation. |

# IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265