User's Guide
SPRUIG6C–January 2018–Revised August 2019

C7000 Host Emulation

C7000™ Host Emulation lets you use C7000 compiler intrinsics and native 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.

Contents

Copyright © 2018–2019, Texas Instruments Incorporated

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 will provide related information for the C7x:

- *C7000 C/C++ Optimizing Compiler User's Guide* (SPRUIG8)
- *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)

1.2 Disclaimer

The material documented in this guide is subject to modification until such time as a formal, supported release of the compiler is complete. This includes names of intrinsic interfaces and processes for leveraging key C7000 compiler features. It is expected that most changes will be driven by internal and external customer feedback.

1.3 Trademarks

C7000, C6000 are trademarks of Texas Instruments.
OpenCL is a trademark of Apple Inc. used with permission by Khronos.
Linux is a registered trademark of Linus Torvalds in the U.S. and other countries.
Windows, Visual Studio are registered trademarks of Microsoft.

Copyright © 2018–2019, Texas Instruments Incorporated

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:

- **Linux** (x86-64 bit)
  - GNU g++ compiler version 5.4.0 or higher
- **Microsoft Windows®** (x86-64 bit)
  - Visual C++ 2015/2017 build tools (standalone or packaged with corresponding Visual Studio® IDE installation)
  - GNU g++ compiler version 6.3.0 or higher (MinGW)

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.

2.3 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:

- 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 with intrinsics. For example, intrinsics that operate directly on memory and the L1D cache cannot be used with C7000 Host Emulation. (See Section 4.5.)
- All source files must #include the `c7x_host_emulation.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.)

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.

Copyright © 2018–2019, Texas Instruments Incorporated

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 will need to #include the `c7x_host_emulation.h` file. This is the only header file required to utilize the C7000 Host Emulation package. It will in turn include all other required header files. For reference, the entire list of header files included with the Host Emulation package can be found in Table 1.

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. For example:

```
#ifdef __C7X_HOSTEM__
#include "c7x_host_emulation.h"
#else
#include "c7x.h"
#endif
```

Table 1. Host Emulation Header Files

File Included Explicitly	**Description**
c7x_host_emulation.h	Main header file, includes all others listed below
Files Included Automatically	
c6x_he_migration.h	Legacy intrinsics and data types
c7x_he_cr.h	Global control register definitions
c7x_he_ecr.h	Global extended register definitions
c7x_he_intrinsics.h	OpenCL-like C7000 intrinsics
c7x_he_load_stores.h	Special load and store intrinsic definitions
c7x_he_luthist.h	Lookup table and histogram control interface
c7x_he_strm.h	Streaming engine control interface
c7x_he_strm_struct.h	Streaming engine definitions
vector.h	Native vector implementation
vector_funcs.h	Helper functions used with vector types within intrinsic definitions

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.

Copyright © 2018–2019, Texas Instruments Incorporated

A full list of C7000 compiler symbols that are defined automatically when using Host Emulation are provided in Table 2

Table 2. C7000 Preprocessor Symbols

Defined Preprocessor Symbols	Description
__C7000__	Defined if compiled for C7000 target or C7000 Host Emulation.
__little_endian__	Defined by default.

__C7X_HOSTEM__ should be defined on the command line when compiling for Host Emulation. It is not 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 */
#ifdef __C7X_HOSTEM__
#include "c7x_host_emulation.h"
#else
#include "c7x.h"
#endif

extern void test(int8 v);

int main()
{
    #ifdef __C7X_HOSTEM__
    int8 vec1 = int8(1,2,3,4,5,6,7,8);

    #else
    int8 vec1 = (int8)(1,2,3,4,5,6,7,8);

    #endif
    int8 vec2 = (int8)(5);
    test(vec1 + vec2);
}
```

Host Emulation compiler command (Linux):

```
g++ -c --std=c++14 -D__C7X_HOSTEM__ -I<cgt_install_path>/host_emulation/include test.cpp
<cgt_install_path>/libc70-host-emulation.lib
```

C7000 compiler command:

```
cl7x test.cpp
```


Copyright © 2018–2019, Texas Instruments Incorporated

4 Intrinsics

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

- OpenCL-Like intrinsics (see Section 4.1)
- Intrinsics used for special loading and storing of vector and scalar elements (see Section 4.2)
- Intrinsics used to program the streaming engine and streaming address generator (see Section 4.3)
- Intrinsics used to migrate legacy code written for the C6000™ compiler (cl6x) (see Section 4.4)
- Intrinsics that act on the memory system (see Section 4.5 for differences)
- Low-level direct-mapped intrinsics (same as C7000 compiler)
- Intrinsics that are a part of the vector predicate to register interface (same as C7000 compiler)
- Intrinsics used to perform lookup table and histogram operations (same as C7000 compiler)

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. The initial value of every control register is set to 0.

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 and may not 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 and any legal use of an OpenCL-like intrinsic is also legal in C7000 Host Emulation.

Copyright © 2018–2019, Texas Instruments Incorporated

4.2 Load and Store Intrinsics

All scalar and vector load and store intrinsics that are available with the C7000 compiler are available for use in C7000 Host Emulation.

While the interface remains exactly the same as it is in the C7000 compiler, error messages may be difficult to decipher as each intrinsic uses a template with many parameters. If an incorrect data type or an incorrect number of elements is used with an intrinsic, then the resulting host compiler error states that "a template substitution error has occurred." This indicates there is no definition for that intrinsic that uses a matching combination of parameters. The following code contains an example of such a substitution error.

```
/* load_store_error_output.cpp */

#ifndef __C7X_HOSTEM__
void print(long* ptr, int length)
{
    /* Implementation is omitted */
}
#endif

#ifdef __C7X_HOSTEM__
// Host Emulation Code
char32 invalid_input = char32(char16(0), char16(1));
__vload_deinterleave_long(&invalid_input).print();

#else
// Target Code
char32 invalid_input = (char32)((char16)(0), (char16)(1));
long8 res = __vload_deinterleave_long(&invalid_input);
print((long*)(&res), 8);
#endif
```

Host Emulation Output (using `g++-5 -std=c++14 load_store_error_output.cpp`)**:**

```
error: no matching function for call to '__vload_deinterleave_long(char32*)'
 __vload_deinterleave_long(&invalid_input).print();

                                         ^
In file included from include/c7x_host_emulation.h:11:0,
                 from spls_error_test.cpp:2:
include/src/c7x_he_load_stores.h:414:25: note: candidate: template<class ELEM_T_IN, long unsigned
int NELEM, class, class> _c70_he_detail::vtype<long int, (NELEM / 2)>
__vload_deinterleave_long(_c70_he_detail::accessible<ELEM_T, NELEM>*)
 vtype<int64_t, NELEM/2> __vload_deinterleave_long(accessible<ELEM_T_IN, NELEM>* input)

                         ^
include/src/c7x_he_load_stores.h:414:25: note:   template argument deduction/substitution failed:
include/src/c7x_he_load_stores.h:413:10: error: no type named 'type' in 'struct
std::enable_if<false, void>'
          typename = typename std::enable_if< (NELEM <= 16) && (NELEM >= 4) >::type>
```

Target Output:

```
"spls_error_test.cpp", line 42: error: (OpenCL) Cannot find overloaded instance for function:
__vload_deinterleave_long
```

4.3 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.

Section 6.1 details implementation requirements for using the streaming address generator with C7000 Host Emulation.

Copyright © 2018–2019, Texas Instruments Incorporated

4.4 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.

4.5 Memory System Intrinsics

The intrinsics listed in Table 3 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 3. 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

Copyright © 2018–2019, Texas Instruments Incorporated

5 Native Vector Types

The C7000 Host Emulation package generally allows for the use of native vector types (for example, `int16`) to be used in the same way as with the C7000 compiler. 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 native vector type feature is not mentioned here but is permissible with the C7000 compiler, the feature *is permissible* with C7000 Host Emulation.

5.1 Constructors

The native vector constructor syntax with parenthesis around the data type is supported only for the C7000 compiler. Using this syntax for C7000 Host Emulation will not cause compile errors, but will cause unexpected results. Therefore, when using C7000 Host Emulation, native vector type constructors must be used without parentheses around the data type itself. Otherwise, the host compiler will treat the operation as a cast and will yield unexpected results.

This following example shows these differences.

```
/* Host Emulation vector constructor syntax examples */

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)
long8 ex5 = (long8)(1,2,3,4,5,6,7,8);       // -> (8,8,8,8,8,8,8,8)
                                            // Valid syntax, but result is unexpected.

/* C7000 compiler vector constructor syntax examples */

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)
long8 ex5 = (long8)(1,2,3,4,5,6,7,8);       // -> (1,2,3,4,5,6,7,8)
```

As a consequence of C++'s casting rules, the native vector constructor syntax with parentheses around the data type can be used when initializing a vector with one value. This is due to the fact that native vector types duplicate a value to all lanes if only one value is provided to the constructor.

The following example is valid for both the Host Emulation and the C7000 compilers:

```
/* Case with only one element provided to constructor */

/* Host Emulation code */
int4 example = (int4)(1); // -> (1,1,1,1)
                          // 1. Cast element "1" to an int4 vector typedef
                          // 2. Call int4 constructor with "1" as only value
                          // 3. Create int4
                          // 4. Same result as int4(1)
```


Copyright © 2018–2019, Texas Instruments Incorporated

5.2 Accessors

C7000 Host Emulation support is provided for all native vector type accessors except for "swizzle" accessors (.sxz, .s0123 etc.). This is due to the fact that there are too many possible combinations of the swizzle accessors and it would not be possible to have definitions for all of them. A workaround is to use a combination of other accessors. The following example shows a workaround in a specific case.

```
/* 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:
// int8 swizzle = ex.s048c159d

// Potential workaround:
int8 swizzle = int8(ex.even.even, ex.odd.even);
```

5.3 Vector Operators

All vector operators are supported when using C7000 Host Emulation except for the ternary operator when vectors are used as the Boolean expression. However, the ternary operator can be used with vector types as long as the Boolean expression is a scalar value. The following example shows this limitation.

```
/* Ternary operator limitation in Host Emulation code */

/* Legal use of ternary operator, bool_expr is scalar */

int32_t bool_expr = 0;

long8 valid_res = bool_expr ? long8(1,2,3,4,5,6,7,8)
                            : long8(2);
// Operation is legal and result is correct
// valid_res = (2,2,2,2,2,2,2,2)

/* Illegal use of ternary operator, bool_vec is a vector */

int8 bool_vec = int8(-1);

long8 invalid_res = bool_vec ? long8(1,2,3,4,5,6,7,8)
                             : long8(2);
// Operation is illegal
// Vector value bool_vec cannot be converted to a boolean value
```

All other operator implementations follow the specification detailed in the OpenCL specification. Illegal uses of an operator will result in compiler errors. However, the type of message received may vary. In a few cases, illegal uses of some operators will result in assertion errors at compile time rather than traditional compiler errors.

5.4 Vector Pointer and Storage Limitations

The C7000 compiler allows references to consecutive data in memory using a pointer to a native vector type. This allows vector operations and vector accesses on data in memory without a vector variable. In addition, the compiler allows code to treat arrays as vectors and vice versa.

However, in C7000 Host Emulation, the vector class contains more than just the data it represents, making its size larger than just the size of its data. As a result, a pointer to an array cannot be directly cast to a native vector type pointer in C7000 Host Emulation.

Example 1 shows code that is correct when used with the C7000 compiler but will cause a run-time error when used with C7000 Host Emulation.

Copyright © 2018–2019, Texas Instruments Incorporated

Example 1. Code for Use with C7000 Compiler

```
/* Load into vector from scalar array in memory */
int array[] = {1,2,3,4,5,6,7,8};
int8 temp = *(int8*)(&array);
    // temp is now an int8 vector with the same data as "array"

/* Store vector as an array in memory */
int array[] = {1,2,3,4,5,6,7,8};
int8 temp = int8(50);
*(int8*)(&array) = temp;
    // "array" now contains the eight elements from vector temp

/* Use vector pointer to modify data in memory */
int array[] = {1,2,3,4,5,6,7,8};
int8* temp = (int8*)(&array);
    // temp now points to data in "array"
    // Native vector type operations are now valid on data in "array"
(*temp).s0 = -1;
    // Modifies array[0];
int4 temp_even = (*temp).even;
    // Grabs even indices of "array" and creates int4 vector

/* Reference members of vector using pointer */
int8 temp = int8(50);
int32_t* ptr = (int32_t*)(&temp);
*(ptr + 1) = -1;
    // now temp.s1 = -1
```

Instead, the C7000 Host Emulation provides special pointer types for vectors and complex element types that allows for pointer casting and basic pointer arithmetic. For example, `uchar64_ptr` is a pointer type that points to a `uchar64` vector. Likewise, `clong_ptr` is a pointer type that points to a `clong`. These pointer types model C++ smart pointers that manage memory in a special way to ensure that the corresponding allocated objects do not leak. The types of C-style casting supported are listed in Section 5.8.

In Example 2, the previous code has been modified for use with Host Emulation. Because this code can also be used with the C7000 compiler, it is the recommended way to handle vector pointers.

Example 2. Code for Use with Host Emulation and C7000 Compiler

```
/* Load into vector from scalar array in memory */
int array[] = {1,2,3,4,5,6,7,8};
int8 temp = *(int8_ptr)array;
    // Equivalent to: temp = *(int8*)(array) on cl7x
    // temp now contains: (1,2,3,4,5,6,7,8)

/* Store vector as an array in memory */
int array[] = {1,2,3,4,5,6,7,8};
int8 temp = int8(50);
*(int8_ptr)array = temp;
    // Equivalent to: *(int8*)(array) = temp on cl7x
    // array now contains: {50,50,50,50,50,50,50,50}

/* Use vector pointer to modify data in memory */
int array[] = {1,2,3,4,5,6,7,8};
int8_ptr temp = (int8_ptr)array;
    // Equivalent to: temp = (int8*)(array) on cl7x
    // temp now points to data in "array"

    // Native vector type operations are now valid on data in "array"
(*temp).s0 = -1;
    // Modifies array[0] using vector accessor
```


Copyright © 2018–2019, Texas Instruments Incorporated

Example 2. Code for Use with Host Emulation and C7000 Compiler (continued)

```
int4 temp_even = (*temp).even;
    // Grabs even indices of "array" and creates int4 vector

/* Reference members of vector using pointer */
int8 temp = int8(50);
int32_t* ptr = (int32_t*)(&temp);
    // Equivalent to: ptr = (int32_t*)(&temp) on cl7x

*(ptr + 1) = -1;
    // now temp.s1 = -1

int32_t data = *((int32_t*)&temp + 1);
    // data = temp.s1
```

When using Host Emulation, it is required that you use the vector and complex pointer types. (see Section 5.8). This enables the host compiler to provide feedback whenever pointers are used in an incorrect or unsupported way. If you do not use these pointer types, the host compiler will still emit errors if you attempt to use any memory intrinsics that require these pointer types. However, if you do not use any intrinsics, the host compiler may not emit any warnings or errors, and you will see runtime errors instead of compile-time errors.

A full comparison of syntax discrepancies between the C7000 compiler and C7000 Host Emulation is covered in Section 10.

5.5 Print Debug Function

A print function is provided with C7000 Host Emulation that can be used on any native 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: (0,0,0,0,0,1,1,1,1)
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
example.lo.lo.lo.print();   // Illegal, result of acccessors is scalar value

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

#else
print((int*)(&example));
print((int*)(&example.lo)); // Output depends on print() implementation
#endif
```


Copyright © 2018–2019, Texas Instruments Incorporated

5.6 Complex Vector Types

All of the limitations that exist with the construction and use of native vector types extend to complex vector types as well. Complex vector types are simply native vectors in which the element type is a complex element. All valid operators, accessors and uses of complex vectors on the C7000 compiler are available for use in the same way when using C7000 Host Emulation, with the exception of the limitations outlined in this document.

There are some extra limitations when using vector to memory conversion intrinsics with complex vectors.

- There is limited support for converting a complex vector pointer to a pointer of its complex component type. For example, a `cint8_ptr` can only be converted to an `int*` if it the `cint8` to which it points was initialized using an `int*` or converted from an `int*`. if, however, the `cint8` to which it points was *not* initialized using an `int*`, then the host emulation framework throws an exception if you attempt to convert it to an `int*`.

```
/* Cannot convert complex vector pointer to complex component type pointer */

cint2 temp = cint2(1,2,3,4);
int* ptr   = *(int*)&temp; // Not allowed. Framework throws a runtime error exception.
*ptr = 50;                 // Undefined
```

- An additional limitation involves casting to a complex vector type from a pointer type that is different than the complex element type and is different than the complex component type. In this case, the type of the pointer passed must be explicitly cast as the same type of the complex component type. If not, the host compiler emits an error.

```
/* Assumption is made on passed pointer type */

int8_t data[] = {1,2,3,4,5,6,7,8};
cint2 vec = *(char2_ptr)(char*)data;  // Invalid. Must be "(int*)data" to work.

cint data_cint[] = {cint(1,2), cint(2,1)};
cint2 vec_cint = *(cint2_ptr)(char*)(&data_cint[0]));
    // &data_cint[0] is cast to a char*
    // Invalid. Must be "(int*)(&data_cint[0])" to work.
```

There are a few extra valid use-cases that the vector to memory intrinsics provide for complex vectors that exist in addition to the use-cases outlined in Section 5.4.The code that follows shows additional examples of legal uses of the vector memory intrinsics with respect to complex vector types in C7000 Host Emulation.

```
/* Extra use-cases that are available when using memory intrinsics with complex vectors */

/* Convert vector pointer to complex element pointer */
cint4 vec = cint4(1,2,3,4,5,6,7,8);
cint_ptr = (cint_ptr)&vec;                         // Valid
cint element = *ptr;                                  // Valid, element = cint(1,2)

/* Convert complex element pointer to vector pointer */
clong data[] = {clong(1,2), clong(1,2), clong(1,2), clong(1,2)};
clong4 vec = *(clong4_ptr)data;           // Valid, vec is filled with elements of "data"

/* Convert complex element component type pointer to vector pointer */
int64_t data_component[] = {1,2,3,4,5,6,7,8};
clong4_ptr vec = (clong4_ptr)data_component;
    // Valid, *vec is filled with complex elements whose
    // real and imaginary components are elements of data_component
    // i.e. &(*vec).s0.r == &data_component[0]
```


Copyright © 2018–2019, Texas Instruments Incorporated

5.7 Complex Element Types

In general, complex element types are used in the same ways with C7000 Host Emulation as they are with the C7000 compiler. However, due to the complexity of their implementation, complex element types cannot be treated as simple containers of their real and imaginary components. Similar to native vector types, this constraint requires the use of complex element pointer types that can be used in pointer casting and basic pointer arithmetic. See Section 5.8 for a list of the forms of C-style casting supported using these pointer types

5.8 Vector and Complex Element Pointer Types

The Host Emulation framework defines a set of vector and complex element pointer types. You must use these types to manage pointers to vectors and complex element objects. The pointer types manage shared pointers that control ownership and allocation of their corresponding object.

Each vector type and complex element type has a corresponding pointer type that is named `<vector_type>_ptr` or `<complex_type>_ptr`. For example:

`uint8_ptr` points to a `uint8` vector.

`clong_ptr` points to a `clong` complex element.

`cint4_ptr` points to a `cint4` complex vector.

The following describes the behavior defined for these pointer types:

- **Creation** based on existing vector or complex element using the `&` address-of operator:
```
ulong2 vect  = ulong2(2, 4);
ulong2_ptr p = &vect;
```
- **Casting** from a scalar pointer:
```
int array[] = {1, 2, 3, 4}
int4_ptr p  = (int4_ptr)array;
```
- **Pointer dereferencing** using the `*` pointer operator. This returns the vector to which the pointer points
```
ulong2 vect      = ulong2(2, 4);
ulong2_ptr vectp = &vect;
ulong2 new_vect = *vectp;
```
- **Pointer arrow** `->` operator. (This is not supported by the C7000 target compiler.)
```
ulong2 vect      = ulong2(2, 4);
ulong2_ptr vectp = &vect;
vectp->s0        = 5;
```
- **Basic Pointer Arithmetic.** If the pointer type is created based on conversion from a scalar pointer to memory, the following pointer arithmetic operations are supported:
  - Post-increment "++"
  - Pre-increment "++"
  - Post-decrement "--"
  - Pre-decrement "--"
  - Plus "+ offset"
  - Minus "- offset"
  - Plus-assignment "+="
  - Minus-assignment "-="

  If pointer arithmetic is attempted on a pointer type not created based on a conversion from a scalar pointer to memory, an exception will be thrown.

Copyright © 2018–2019, Texas Instruments Incorporated

- **C-style casting**
  - **Converting from a vector**
    - Convert from a vector pointer to an element type scalar pointer.
      ```
      // Converts int4_ptr to int32_t*
      int32_t *p = (int32_t*)pointer_to_int4;
      ```
    - Convert from a complex vector pointer to a complex element type pointer.
      ```
      // Converts cint4_ptr to cint_ptr
      cint_ptr p = (cint_ptr)pointer_to_cint4;
      ```
    - Convert from a complex vector pointer to a complex element type component scalar pointer. This is allowed only if the complex vector was converted/initialized based on a scalar pointer to memory.
      ```
      // Converts cint4_ptr to int32_t*
      int32_t *p = (int32_t)pointer_to_cint4;
      ```
  - **Converting to a vector**
    - Convert from a scalar pointer to a vector pointer.
      ```
      // Converts int32_t* to int4_ptr
      int4_ptr p = (int4_ptr)pointer_to_int32_t;
      ```
    - Convert from a scalar pointer to a complex vector pointer.
      ```
      // Converts int32_t* to cint4_ptr
      cint4_ptr p = (cint4_ptr)pointer_to_int32_t;
      ```
    - Convert from a complex element type pointer to a complex vector pointer.
      ```
      // Converts cint[] to cint4_ptr
      cint data[] = {cint(1,2), cint(1,2), cint(1,2), cint(1,2)};
      cint4_ptr p = (cint4_ptr)data;
      ```
  - Convert from complex element pointer to scalar pointer
    ```
    // Converts cint_ptr to int32_t
    int32_t *p = (int32_t*)pointer_to_cint;
    ```
  - Convert from scalar pointer to complex element pointer
    ```
    // Converts int32_t* to cint_ptr
    cint_ptr p = (cint_ptr)pointer_to_int32_t;
    ```

NOTE: The formerly-documented vtos_ptr(), stov_ptr(), ctos_ptr(), and stoc_ptr() intrinsics are still available, but are deprecated. They now wrap the corresponding C-style cast describedabove.

No other operations are supported.

The C7000 Compiler supports all of these pointer types, which are typedefs to a standard pointer to the corresponding vector or complex element type.

NOTE: A __STRM_TEMPLATE_ptr type is provided for use when storing and loading a __STRM_TEMPLATE. The stream template is used to configure the Streaming Engine and Streaming Address Generator (see Section 6). A `strmtemplate_ptr(addr)` conversion macro is also provided to allow you to cast a scalar pointer to a __STRM_TEMPLATE_ptr type.

Copyright © 2018–2019, Texas Instruments Incorporated

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.

6.1 Streaming Address Generator

As discussed in Section 5.4, when loading and storing vectors from memory, care must be taken to ensure that the data is formatted in the correct way and that the size of the data matches its destination.

When using the Streaming Address Generator, all pointers used as the base address to the SA intrinsics must point to a contiguous set of elements in memory. The base pointer in these intrinsics cannot be a vector pointer if the data at that location is a vector type that was stored directly without modification. To store pre-built vector types into memory for use with the SA, use the scalar-pointer to vector-pointer typecasts described in Section 5.8.

Although the SA can only retrieve offsets to a set of consecutive elements in memory, vector operations are still valid on the data as if it was originally stored as a native vector type. When using the SA to retrieve an offset to a vector type, a pointer is returned to a vector that represents that continuous data in memory. Modifying this vector through the pointer modifies the data in memory, as is expected. The following example demonstrates this limitation and shows how to work with the SA using C7000 Host Emulation.

```
/* SA example: Host Emulation Code */

#include "c7x_host_emulation.h"

int32_t mem[16] = {0};

void SA0_init_func()
{
    // SA initialization of param vector omitted

    __SA0_OPEN(param_vec);
}

int main()
{
    int16 to_mem = int16(int8(0), int4(1), int4(2));
    *(int16_ptr)(mem) = to_mem;
    // mem now contains
    // {0,0,0,0,0,0,0,0,1,1,1,1,2,2,2,2}

    SA0_init_func();

    int16* data = __SA0ADV(int16, mem);
    // It is also valid to use:
    // int16* data = __SA0ADV(int16, (int32_t*)mem)
    // This is because "mem" points to a list of consecutive elements

    (*data).s0 = -1;    // Modify mem using native vector type
    // mem now contains
    // {-1,0,0,0,0,0,0,0,1,1,1,1,2,2,2,2}
}
```

As was noted in Section 5.4, the vector pointer to scalar functions are defined with the C7000 compiler as a simple cast operation. This allows any code written with the SA under C7000 Host Emulation to compile properly with the C7000 compiler without modification.

Copyright © 2018–2019, Texas Instruments Incorporated

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.

Copyright © 2018–2019, Texas Instruments Incorporated

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 4.5, 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 4 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 4. 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

Copyright © 2018–2019, Texas Instruments Incorporated

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_host_emulation.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 on Host Emulation
    // Note: this code is legal on C7000
    __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);

    // Legal syntax, but not recommended
    // Code will not compile on C7000
    float lo = src1.get_lo();

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

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


Copyright © 2018–2019, Texas Instruments Incorporated

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.

Copyright © 2018–2019, Texas Instruments Incorporated

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 native vector constructors, compiler errors and warnings may reference different classes and their respective members. Table 5 lists these key terms and their purposes.

Table 5. : 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>(0)'…"
Vtype	High-level vector class name	"Error: conversion from 'int2 {aka _c70_he_detail::vtype<int, 2ul>]'…"
Accessible	Class name that represents an "accessible" level of a vector (i.e. vec.lo)	"Error: char32 is not derived from '_c70_he_detail::accessible<char, 16ul>'…"

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 6 lists the host compiler errors and warnings, or lack thereof, which may arise when using the original C7000 syntax with C7000 Host Emulation.

Table 6. Syntax change related compile errors and warnings

Description	Example	Compiler Output

Using C7000 vector constructor syntax with Host Emulation	(long8)(1,2,3,4,5,6,7,8) // C7000
	vs.
	long8(1,2,3,4,5,6,7,8) // Host Emu
		No errors or warnings. Results are incorrect.

Ternary operator with vector as "boolean expression"	res = vec1 ? vec2 : vec3	Compiler error: "Cannot convert vec_type to bool".
Using swizzle accessor	example.s0121	Compiler error: "Member does not exist".

Using actual vector/complex pointers in casting operation rather than the special pointer types described in Section 5.8.	int4 vect = *(int4*)data; // C7000
	int4 *vp = (int4*)data; // C7000
	vs.
	int4 vect = *(int4_ptr)data; // Host Emu
	int4_ptr vp = (int4_ptr)data; // Host Emu
		Run time error.

Using invalid value within SE/SA parameters	Setting VECLEN to a negative number.	Run time error. Explains which flag is invalid and what constraints it must meet.

Copyright © 2018–2019, Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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, 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 (www.ti.com/legal/termsofsale.html) 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.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2019, Texas Instruments Incorporated