

Introduction

This guide provides assistance for programmers porting their code from release 2.5 to release 3.x of the ClearSpeed SDK. There are a number of new features and changes to the way some existing features are implemented. Some changes are to provide support for new hardware products or software features planned for future releases.

The first section provides an overview of the changes and new features. The second section provides detail on the changes that you may need to make in order for your code to compile or execute correctly. The third section describes new features that you may wish to take advantage of.

Refer to the *SDK Reference Manual* for more information about the features discussed in this document.

1 New features

The following changes and new features are discussed in this document.

- Dynamic stack support: the Release 3 compiler includes support for dynamically allocated stacks, run time stack checking and a new mechanism for specifying the size of statically allocated stacks.
- Changes to the Vector Math Library (VML).
- Reorganization of Cⁿ and assembler header files.
- Reduction operations in a new Cⁿ library.
- Operator overloading simplifies the use of VECTOR data types in expressions.
- Reorganization of the microcoded instruction set. A number of infrequently used instructions have been moved from the core instruction set to an extension library.
- New predefined macros in the compiler
- Support for future ClearSpeed processors with error correcting (ECC) memory.

2 Required changes

The following sections describe changes that *must* be made to software that uses the following:

- User defined sizes for mono or poly stacks
- Reorganization of Cⁿ and assembler header files.
- Assembly code using instructions moved into the extension set

2.1 Stack size definition

Warning: If you specify stack sizes in your code then you *must* change your code to use one of the new methods. Failure to do so may cause your program to fail in unpredictable ways.

Note: If you get an unexpected “Not enough memory” error from `csrrun` or `CSAPIload` then a possible cause is the use of the old-style stack size specification.

By default, the mono and poly stacks for the main execution thread (thread 0) are dynamically sized, up to the maximum available memory. If a size is specified for these stacks then they will become static, with a fixed size. See section [3.1: Dynamic stack support on page 5](#) for more information.

Note: In many cases, the use of a dynamic stack for the main execution thread in the 3.0 release may remove the need to explicitly specify the size of the stacks.

In the previous SDK releases, stacks had to be created by defining an appropriate symbol, typically by creating an assembler source code file containing the necessary definitions and then linking this with your program.

With this release, the size of statically allocated stacks can be defined in a number of different ways. You can use whichever method is most convenient for your application and development process.

The size of the stacks can be specified in any compilation unit making up an executable. However, any particular stack must only be specified once.

2.1.1 Command-line options

The compiler has a set of command-line options for specifying the sizes of the stacks for each thread. See the *SDK Reference Manual* for details.

The command line options are of the form:

```
set-mono-stack-size-thread-n size
set-poly-stack-size-thread-n size
```

Where *n* is the thread number that a stack is being defined for and *size* is the size of the stack in bytes.

2.1.2 Cⁿ pragmas

A Cⁿ pragma is also provided for setting the stack size, as follows:

```
#pragma static_stack(mono|poly, size, thread)
```

The first argument defines whether a mono or poly stack size is being defined. The second argument specifies the stack size in bytes. The third argument specifies the thread number that the stack is for.

2.1.3 Assembly code

Note: The other methods of specifying the stack size all map on to this underlying mechanism by causing the compiler to emit the necessary directives to define the stacks.

The mechanism for specifying the size of stacks in assembler code has changed. Instead of defining the stacks in the mono and poly bss sections using the `.fill` command, you now use the `.set` directive to specify the size of the stack you need. The runtime allocates this space when the executable is loaded.

For example, previously you would have included code similar to the following fragment in your program:

```
.section .mono.bss
__FRAME_BEGIN_MONO__3::
.global __FRAME_BEGIN_MONO__3
.fill 1024, 1, 0x0
```

This creates a 1 KB mono stack frame for thread 3 in the mono bss section.

The new method of declaring this is to use a `.set` command as shown below:

```
.set __FRAME_BEGIN_MONO__3, 1024
.global __FRAME_BEGIN_MONO__3
```

This will cause the runtime to allocate an area of this size for the stack at load time and assign all necessary pointers to this point. All stacks, both mono and poly, can be declared in this manner.

Note: The same symbols are used to define the stack sizes as were previously used to define the start address of the stack. Therefore, your program will fail to run correctly if you do not change, or remove, the old code for specifying stack size.

2.2 Reorganization of header files

The directory structure for Cⁿ and assembler code header files has been rationalized to separate those used on the host and the card. If you just use the setup scripts and shortcuts installed with the software then you should not need to do anything: the search paths will be updated and your code should just compile as before.

If you use your own initialization environment setup via scripts or within an IDE for example, then you will need to update the paths.

New `card` and `host` subdirectories have been created under the `include` directory.

The `card` directory has a `cn` subdirectory for all the standard Cⁿ header files and an `asm` subdirectory for all the assembler include files (including those for the card-side FFT library).

The `host` directory contains the header files needed by host-side code, for example the CSAPI header file (`csaspi.h`) and the host-side FFT headers.

2.2.1 Removed files

The following files have been removed from the Cⁿ include files as they contain no information relevant to the CSX architecture or libraries.

```
_ansip.h
_syslist.h
alloca.h
fcntl.h
floatp.h
ieeefpp.h
limitsp.h
localep.h
newlib.h
newlibp.h
signal.h
stdargp.h
time.h
unistdp.h
host/ev5_regmap.h
host/cs_sys/Endian.h
host/cs_sys/types.h
machine/stdlib.h
machine//time.h
machine/types.h
sys/_typesp.h
sys/cdefs.h
sys/configp.h
sys/fcntl.h
sys/features.h
sys/lockp.h
sys/signal.h
sys/stat.h
sys/stdio.h
sys/stdiop.h
sys/typesp.h
sys/unistd.h
sys/unistdp.h
```

2.3 Instruction set changes

The poly instructions for the CSX600 are implemented in microcode. To make more effective use of the available microcode store a number of instructions have been moved out of the base instruction set. These are rarely used and application specific instructions which are not used by the compiler. Some of these instructions are now available in a new instruction set extension library.

2.3.1 Instruction set extension library

The instructions described in the “complex instructions” section of the *Instruction Set Reference Manual* and a number of more complex “swizzle” instructions have been moved to the new extension library. Instructions in the extension library are identified by a comment

in the appropriate instruction definition found in this manual. If you use these instructions in your program, you will need to make the following changes:

1. Add a call to the function `__cn_extension_ucose()` to your program to initialize the instruction set extension library. This must be done before using any of the instructions in this set.
2. When compiling your program, use the command line option with `cscn`:
`--use-add-ucose cn_extension_ucose`
This informs the assembler and linker about the new instructions in the library.

2.3.2 FFT instructions

The Fast Fourier Transform instructions used by the CSDFT library have been removed from the standard instruction set. These instructions are loaded directly by the CSDFT library when it initializes.

Note: It is possible to use both the instruction set extension library and the CSDFT library simultaneously although this is not currently documented. Contact ClearSpeed support (via the support website <http://support.clearspeed.com>) if you need more information.

3 Exploiting new features in SDK 3.0

3.1 Dynamic stack support

The way that stacks are handled by the SDK has been improved to provide more flexibility and the ability to check stack usage at run time.

Each thread executing in a program will probably require a mono and a poly stack. In previous releases, all stacks were statically sized.

In the 3.0 release, stacks are automatically created for the main execution thread (thread 0) and these stacks are dynamically sized. As the program runs, more stack space will be used, up to the limit of available memory. Run-time stack checking can be enabled to ensure that the stack usage does not exceed available memory.

Thread 7 is used by the asynchronous memcopy functions. These functions allocate static stacks of size zero for this thread.

If you use any other threads in your program then you must allocate mono and poly stacks for each thread. Normally, these stacks will be assigned fixed sizes and allocated statically (see [2.1: Stack size definition on page 2](#)).

One mono stack and one poly stack can be defined to be managed dynamically. By default, these are allocated to thread 0. If you wish to assign a dynamic stack to another thread then you must first assign a static stack for thread 0. To define a dynamic stack for a thread, you should set the stack size to -1. This will convert the stack to a dynamic stack. If you have two threads with dynamic stacks in the same domain (that is, two mono or two poly dynamic stacks) this will cause a run-time error.

3.2 Stack checking

This release also includes support for run-time checking of stack usage. There are two types of checks that can be performed: a static, compile time, check on the maximum size of any functions stack frame; secondly, a run-time check that the program does not run past the end of the available memory.

3.2.1 Run-time stack checking

The option `dynamic-stack-check` turns on run-time checking of mono and poly stack usage for all threads. This causes the compiler to add code the entry point of every function to check that the function's stack frames will not cause the stacks to overrun available memory. This can have a significant performance penalty and so is turned off by default. This option should only be used when developing or debugging software.

For statically allocated stacks, this checks that the mono and poly stack frames for a function will not exceed the size specified for the stack.

For dynamically allocated stacks, this checks that the mono and poly stack frames for a function will not cause the stack to grow beyond the available memory. For the mono stack this means that the stack will not overrun the heap (which grows from the top of memory). For the poly stack, it checks that the stack will not grow past the end of poly memory.

If these checks fail then the runtime will display a warning on the console when that function is called.

3.2.2 Stack frame size checking

The compiler can also check (at compile time) that the stack frame for any function does not exceed a given size. This is a simple test that does affect performance and gives some confidence that stack usage is not grossly excessive. This check can be turned on using the compiler command line options `check-mono-frame` and `check-poly-frame`.

The default limits for stack frame sizes are 64 KB for mono and 3 KB for poly. These limits can be changed with the command line options `check-mono-frame-size` and `check-poly-frame-size`.

If this check fails, then the compiler will generate a warning. You can cause the compiler to report an error when the check fails by using the `error-mono-frame` or `error-poly-frame` options.

3.3 Operator overloading

The compiler now supports operator overloading allowing several of the standard arithmetic operators to be used with the vector data types in the Cⁿ language. For example, instead of an expression of the form:

```
vrs = __cs_vadd(v0, v1);
```

It is now possible to write:

```
vrs = v0 + v1;
```

This greatly simplifies writing complex expressions involving vector operands.

3.4 Changes to standard libraries

3.4.1 Vector math library

The microcoded instructions used by the Vector Math Library (VML) are now included in the standard instruction set. Initialization is now done automatically. Therefore, the VML has deprecated the following functions and command line option:

1. The functions `cs_vmathp_init()` and `cs_vmathp_init_sp()` are no longer required.
Calling these functions is no longer necessary and will have no effect. They may be removed in a future release of the software.
2. The command line option `-m cn_vmath_microcode` is no longer required.
Using this command line option will have no effect. The microcode library file still exists but is empty. This file may be removed in a future release.

3.4.2 Reduction library

A new Collectives library provides a set of reduction operations on poly data. The operations supported in this release are: sum, product, minimum and maximum.

See the *Cⁿ Standard Library Reference Manual* for details of these functions.

3.5 Predefined macros

This release includes a new macro to identify the version of the Cⁿ compiler. The macro `__CSCN_VERSION__` contains a string representing the version. See the *SDK Reference Manual* for details. As this is a new macro, this release can be distinguished from earlier ones simply by testing to see if this macro is defined.

3.6 Support for ECC memory

Future ClearSpeed processors will include memory with error correcting codes (ECC) for greater reliability. This applies to both the on-chip SRAM and the poly memory (in addition to the current ECC support for external DRAM). The use of ECC memory introduces some constraints on the way that memory can be accessed. Stores to memory which are the same width as the ECC word are not affected. Stores which are narrower than this need to perform a read-modify-write of the whole word to avoid causing an ECC error. These instruction will therefore be slower in this release. However, this is not expected to have a significant impact on the overall performance of most applications.

Support for ECC memory in future products is being introduced in this release of the SDK to avoid future compatibility changes.

On-chip SRAM memory will have 32-bit ECC and so a simple store must be 32 bits wide. If you wish to write 16-bit data or a single byte to this memory then you will have to read the 32-bit word from memory, modify the bytes to be written and write the whole word back.

Poly memory will have 16-bit ECC. The poly load and store instructions have been modified to perform a read-modify-write cycle for all one-byte stores. All poly one-byte stores now also require additional temporary registers in order to perform this read modify write logic. If you use these instructions in assembly code, you will need to allocate these temporary

registers using the `.setmonotemp` and `.setpolytemp` directives (see the Assembly Language chapter of the *SDK Reference Manual* for more information). [Table 1](#) lists the temporary registers required by each form of the store instruction.

Store type	Sub type	Mono temporary registers	Poly temporary registers	Cycle count	
				Non-ECC	ECC
Store	Immediate address	2	3	2	13
	Immediate address, mono offset	2	3	1	12
	Mono address, no offset	2	3	1	12
	Mono address, with offset	2	3	1	12
	Poly address, no offset	0	5	2	17
	Poly address, with offset	0	5	2	17
Indexed store	Immediate address	0	7	1	17
	Mono address, no offset	0	7	1	16
	Mono address, with offset	0	7	1	18
Forced store	Immediate address	2	4	2	19
	Immediate address, mono offset	2	4	1	19
	Mono address, no offset	2	4	1	18
	Mono address, with offset	2	4	1	19
	Poly address, no offset	0	6	2	23
	Poly address, with offset	0	6	2	23
Forced indexed store	Immediate address	0	8	2	23
	Mono address, no offset	0	8	1	22
	Mono address, with offset	0	8	1	24

Table 1. Temporaries required

If you are writing code which will *only* be run on the CSX600, or another processor without ECC poly memory, then it is possible to make the tools use the old single-byte store instructions. These do not do a read-modify-write and so are faster.

You can force the compiler to use the non-ECC store instructions using the `cscn` command line option `--no-poly-ecc`.

You can make the assembler use the old style store instructions by inserting the following pragma at the top of your assembler source code file.

```
.pragma non_ecc_st on
```

Revision history

Date	Revision	Changes
October 2007	1.0	New document.
November 2007	1.A	More information added after Beta release
January 2008	1.B	New template applied

ClearSpeed Technology, Inc.

800 West El Camino Real
Suite 180
Mountain View, CA 94040

Tel: +1 650 943 2329
Fax: +1 650 962 1188

ClearSpeed Federal Systems, Inc.

228 Hamilton Avenue, 3rd Floor
Palo Alto, CA 94301

Tel: +1 650 798 5027
Fax: +1 650 798 5001

ClearSpeed Technology plc

3110 Great Western Court
Hunts Ground Road
Bristol BS34 8HP
United Kingdom

Tel: +44 (0)117 317 2000
Fax: +44 (0)117 317 2002

Email: info@clearspeed.com

Web: <http://www.clearspeed.com>

Support: <http://support.clearspeed.com>

1. Information and data contained in this document, together with the information contained in any and all associated ClearSpeed documents including without limitation, data sheets, application notes and the like ('Information') is provided in connection with ClearSpeed products and is provided for information only. Quoted figures in the Information, which may be performance, size, cost, power and the like are estimates based upon analysis and simulations of current designs and are liable to change.
2. Such Information does not constitute an offer of, or an invitation by or on behalf of ClearSpeed, or any ClearSpeed affiliate to supply any product or provide any service to any party having access to this Information. Except as provided in ClearSpeed Terms and Conditions of Sale for ClearSpeed products, ClearSpeed assumes no liability whatsoever.
3. ClearSpeed products are not intended for use, whether directly or indirectly, in any medical, life saving and/ or life sustaining systems or applications.
4. The worldwide intellectual property rights in the Information and data contained therein is owned by ClearSpeed. No license whether express or implied either by estoppel or otherwise to any intellectual property rights is granted by this document or otherwise. You may not download, copy, adapt or distribute this Information except with the consent in writing of ClearSpeed.
5. The system vendor remains solely responsible for any and all design, functionality and terms of sale of any product which incorporates a ClearSpeed product including without limitation, product liability, intellectual property infringement, warranty including conformance to specification and or performance.
6. Any condition, warranty or other term which might but for this paragraph have effect between ClearSpeed and you or which would otherwise be implied into or incorporated into the Information (including without limitation, the implied terms of satisfactory quality, merchantability or fitness for purpose), whether by statute, common law or otherwise are hereby excluded.
7. ClearSpeed reserves the right to make changes to the Information or the data contained therein at any time without notice.

© Copyright ClearSpeed Technology plc 2007, 2008. All rights reserved.

Advance is a registered trademark of ClearSpeed Technology plc

ClearSpeed, ClearConnect, Advance and the ClearSpeed logo are trade marks or registered trade marks of ClearSpeed Technology plc. All other brands and names are the property of their respective owners.