

ClearSpeed[™]



Developer Package Release Notes

Software Release 2.51

06-XX-1404 0.0

August 2007

Table of contents

Release Notes for the Developer Package	4
What's new in Release 2.51	4
Issues fixed in Release 2.51	4
Known Developer Issues	4
Installation	5
Compiler	5
Debugger	6
Profiler	6
Assembler	6
Libraries	6
Simulator	8
Instruction Set	9

Release Notes for the Developer Package

This document lists the most important changes to the Software Development Kit (SDK) since release 2.50. In addition, it lists the known open issues and limitations in release 2.51.

For more information regarding the status and workarounds related to any of these issues, please contact ClearSpeed support quoting the relevant CTS number.

You should check the ClearSpeed customer support website (<http://support.clearspeed.com>) for updates to these release notes.

What's new in Release 2.51

This release includes a number of bug fixes since the 2.50 release.

Issues fixed in Release 2.51

The following issues have been fixed in this release.

CTS 1629: There were certain operations that could be done in ordinary assembly code that were not possible to replicate when using vector types in inline assembly.

The first case was as follows (where `0:p16` is a vector):

```
ld 0:p16, 0:m2;
```

Similarly, there was no way to do the following (where `0:p16` and `16:p16` are vector types):

```
mov 0:p8, 16:p8;  
mov 8:p8, 24:p8;
```

CTS 2084: For poly load with offset instructions, the hardware only supported immediate offsets of less than 128 from the mono base address. The instruction set provided a macro for offsets greater than 127, however this wasn't correctly reflected in the instruction set documentation.

CTS 3116: The assembler assumed immediates were four bytes. For moves with immediates larger than four bytes, such as the following example, the assembler should have emitted multiple two-byte moves to make up the 16 bytes.

```
mov 0:m16, 0
```

A bug existed in the handling of the immediate such that this is broken down to zero length values and so the `mov` instructions were not generated correctly. A work around in this case was to explicitly specify the size of the immediate, for example:

```
mov 0:m16, 0:16
```

CTS 2997: If you install the SDK and then uninstall it on Microsoft Windows, some files were deleted which are required by the runtime software. The result of this was that it was no longer be possible to run any software on the board until the runtime software or SDK was reinstalled.

Known Developer Issues

The following issues are currently open.

Installation

CTS 1285: The Windows XP installer for the SDK creates an empty file called `Admin` in the directory it is run from. This file can be deleted or ignored.

CTS 2119: Object files or executables created with a given release of the SDK are not guaranteed to be compatible with other versions of the runtime libraries.

Running code compiled with one version of the SDK with a different version of the runtime may cause errors of the form:

```
source 1:06 destination 0:00 Avci read request pktid:83 trdid:01
plen:010 Addr: 0000
```

When a new version of the runtime and driver software is installed, any existing binaries must be recompiled with the corresponding version of the SDK.

Compiler

CTS 4321: The compiler will currently generate incorrect assembly for floating-point expressions that can be statically evaluated to NaN or Inf. Unless there is a floating-point constant in the source code that is either NaN or Inf, the static evaluation of the expression by the compiler can be suppressed by compiling without optimization (using the `-O0` flag).

CTS 4404: The compiler will fail with an internal error on the declaration of functions that return a struct and that have an empty parameter list. For example the declaration of the function `f` below will cause the error:

```
struct foo {char x, y, z[2];};
struct foo f();
void bar(int baz)
{
f().z[baz] = 1;
}
```

To avoid the error, the user should use the `void` keyword in the function declaration. For example:

```
struct foo f(void);
```

CTS 4407: When initializing global declarations, certain address calculations in the initializer expression may cause internal compiler errors. For example, the code below will not compile:

```
int z = (&"Foobar"[1] - &"Foobar"[0]);
```

As a workaround for this problem, the user should perform the initialization within a function.

CTS 4408: The following piece of code currently causes an internal compiler error.

```
unsigned long x[4];
void foo(void)
{
((void (*)(x+2))());
}
```

If you encounter a compiler error where your code is similar to the above code, compiling without optimization (using `-O0`) should allow compilation to continue.

CTS 4409: Certain function declarations for which the parameters are undefined will cause internal compiler errors. The error generated will look like:

```
Internal fatal error in engine match
Rewrite: unknown operator `mirCast` in domain mirEXPR
```

For example, the following declaration is incomplete:

```
extern void b();
```

If the function accepts no arguments, then a complete declaration would be:

```
extern void b(void);
```

In order to prevent such errors, make sure your function declarations are complete.

Debugger

CTS 309: It is not possible currently to cast a value in the `csgdb` command language to a poly type.

CTS 393: It is not possible to currently view poly structure members individually. The whole structure will be displayed. Trying to reference the individual members of poly structures will produce an error message.

CTS 864: If you read poly registers or poly memory while using `csgdb` to single step through the instructions which issue a PIO transfer, it is possible to corrupt the data in the PIO node.

CTS 949: If a breakpoint is set on the macro `cycles.get`, it is possible the breakpoint will be hit again on continue as the macro may wrap around when the cycle count register wraps.

Using one-shot breakpoints when attempting to break on `get.cycles` will avoid the problem.

CTS 1875: If you attempt to debug code executing on thread 1 when it is currently executing a `sem.wait` instruction, the debugger will lock up if an attempt is made to read poly state.

CTS 4417: The debugger does not work correctly with dynamically linked executables. In particular, it is not possible to step into a function in a dynamically linked program. If this is attempted, the code will run until the next breakpoint or the end of the program. In addition a number of bus error messages may be generated. Also, attempting to disassemble dynamically linked code will disassemble code at the wrong address.

CTS 4482: In some cases `csgdb` will report erroneous values for function parameters. Bus error messages may also be generated. This may vary depending on the line where the breakpoint is set.

Profiler

CTS 3745: The ClearSpeed Visual Profiler tool (`csvprof`) for both Linux and Windows requires that Sun's Java Runtime Environment version 1.5 or later be installed as a prerequisite.

Assembler

CTS 568: The assembler data directives (`.byte`, `.float`, `.int` and `.short`) can be used with a constant expression as the parameter(s). The assembler will not accept an expression as the parameter to the `.double` directive. The parameter to `.double` must be a floating point constant.

Libraries

CTS 647: Calling `dprintf` frequently on the board will cause the host to block other interrupts while each print is completed. This may cause the system to slow down if semaphores are being used to communicate between the host and the board at the same time.

CTS 1257: Currently, the size of both the mono and poly stack sizes are hard coded in the **Cⁿ** runtime. The current defaults are 64 KB for mono and 3 KB for poly. These defaults can be changed by overriding the symbols associated with the stacks. There are two stages to this: assign a new memory size to the stack symbols, then ensure the new symbol values are used by redefining the entry point of the code.

This can be done using either **Cⁿ** code or assembly code. Both methods are described below.

Note: The stacks must be 16-byte aligned.

Setting stack sizes in Cⁿ

To allocate memory for the stacks, you can declare arrays with the same name as the stack symbols, as shown below, in the source file which includes `main()`.

```
#pragma align 16
mono unsigned char __FRAME_BEGIN_MONO__[MONO_STACK_SIZE];

#pragma align 16
poly unsigned char __FRAME_BEGIN_POLY__[POLY_STACK_SIZE];

...

void _start(void)
{
    exit(main());
}
```

In this example, the macros `MONO_STACK_SIZE` and `POLY_STACK_SIZE` define the stack size in bytes.

Setting stack sizes in assembler

To modify the stack sizes using assembler code, place the following in a `.is` file then assemble and link with the rest of the program.

```
.section .mono.bss
.align 16
__FRAME_BEGIN_MONO__:
.fill 64 * 1024 // these are the number to change
.global __FRAME_BEGIN_MONO__

.section .poly.bss
.align 16
__FRAME_BEGIN_POLY__:
.fill 1024 // these are the numbers to change
.global __FRAME_BEGIN_POLY__

.section .text
_start:
.global _start
.type _start, @function

j.sub main;
terminate;
```

In the above example, the numbers after the `.fill` directive specify the size of memory allocated for the stacks, in bytes. In this example they are set to 64 KB for mono and 1 KB for poly.

CTS 2155: All the standard C header files are included in the SDK installation. These may include declarations of functions which are not currently supported. Please refer to the [The Cⁿ Standard Library Reference Manual](#) for information on the supported functions.

CTS 4243: The documentation for the standard libraries contains sections on memory usage for each function. Some of the figures in these tables are incorrect. The columns for mono data size, mono bss size and poly bss size should be ignored. The figures in the column labelled poly data size are, however, correct.

CTS 4322: Calling `printfp` to display a poly floating point expression that evaluates to NaN may cause the code to hang at runtime. If the programmer suspects that this may be the case, recompiling without optimization (`-O0`) may allow the program to continue at runtime, but the result of the `printfp` call will still be incorrect.

CTS 4339: The single precision `cs_invp` function in the vector math library will currently return an incorrect result in certain situations. The problem does not occur in double precision. The workaround for this if you wish to do a vector reciprocal is to do four individual divides or use `cs_isqrtp`.

For example, replace:

```
y = cs_invp(x);
```

with:

```
y = __cs_vpconstructor(1/__cs_vindex(x,0),
                      1/__cs_vindex(x,1),
                      1/__cs_vindex(x,2),
                      1/__cs_vindex(x,3));
```

or:

```
y = cs_isqrtp(x);
y = __cs_vmul(y,y);
```

The second approach is quicker than the first but will give a slightly less accurate result.

CTS 4373: The `sprintf()` function does not append a null (`'\0'`) character to the end of the generated string. For example, the following program:

```
#include

int main(void) {
    char str[64];

    sprintf(str, "Hello world\n");
    printf(str);
    sprintf(str, "Hello");
    printf(str);
    return 0;
}
```

Will generate the output:

```
Hello world
Hello world
```

The workaround for this problem is to add an explicit format character to insert a zero character, for example:

```
sprintf(str, "Hello\n%c", 0);
```

Note: It is not sufficient to simply include '\0' at the end of the format string.

Simulator

CTS 191: When code executing on the simulator writes to address `0x0`, the simulator will stop with an error indicating that it has tried to access out-of-range memory. This causes `csgdb` or `csr` to become unresponsive.

CTS 4070: If `isim` detects an illegal operation (such as a non-aligned memory access) that causes it to display an error message this will cause a memory leak. If the error occurs repeatedly, the error message will only be printed once but more memory will be allocated on each occurrence.

Instruction Set

CTS 2460: There is an error in the swizzle documentation. All of the following instructions can also take eight-byte operands, each taking four cycles:

- `swizzle.low.out.get`
- `swizzle.high.out.get`
- `swizzle.low.in.put`
- `swizzle.high.in.put`

There is also an error in some of the arithmetic instructions. Both `addc` and `subc` support the following versions, the cycle counts being as follows:

- `addc p2, p2, p2` 2 cycles
- `addc p4, p4, p4` 4 cycles
- `subc p2, p2, p2` 2 cycles
- `subc p4, p4, p4` 4 cycles