CodeWarrior for NINTENDO DS 2.0
Transition Notes

Rev. 1.1 2006 April 7

Games Organization, Developer Technology

Freescale Semiconductor, Inc.

Table of Contents

1 Introduction

CodeWarrior for NINTENDO DS 2.0 has a new compiler and linker. This document discusses changes to these tools since version 1.2 that NINTENDO DS programmers should know.

Almost all descriptions in this document are culled from other existing documents which the installer installs and are listed below. Please also refer to them.

\Release_Notes\ARM\Compiler_ Release_ Notes
ARM_Compiler_Notes.txt
ARM_Linker_Notes.txt
CW C-C++ Notes 4.x.txt

In this document, “CodeWarrior for NINTENDO DS” is sometimes referred to “CW for DS.”

1.1 Document Configuration Changes

In CW for DS v2.0, the configuration of documentation has changed since previous version. The C Compilers Reference has been removed in favor of the Build_Tools_ Reference_NINTENDO_DS. In the Targeting Manual, portions describing the build tools, including the linker, have been moved into Build Tools Reference document.

2 Code generation

2.1 New Front End

Release 2.0 of the tools includes a new and improved ARM compiler that features a new front-end (version 4.0) supporting recent changes in the C++ standard and other back-end improvements.

2.2 Using IPA File

Inter-procedural Analysis has been added as part of the CodeWarrior for Nintendo DS v2.0 compiler. For this option, a note on the impact of using of IPA|file: In some cases, transitioning from the v1.2 compiler to the v2.0 compiler will result in larger .data and .rodata sections. Using the compiler optimization "-ipa file" when using the command line compiler, or selecting the "File" option for the IPA setting in the C/C++ Language Preference Panel will turn this optimization on. The Makefiles in the NITRO-SDK 3.1 use -ipa file as the default setting. Stationery files for NITRO-SDK3.0 in this release also use the –ipa file setting by default.

The Interprocedural Analysis switch “ipa” has 3 options: Per-function optimization, Per file optimization and Per-program optimization. However, Per-program IPA mode has not fully-tested for ARM development. Use at your own risk.

2.3 Emission of Variables into Data Section

Previous compilers in CW for DS 1.x emit plural ELF sections for each variable separately. The compiler in DS2.0 emits 1 ELF section for all variables in a compilation unit.

2.3.1 Using __attribute__ ((aligned(n))) to Specify Variable Boundary

The changes to how variables are emitted (section 2.3) affects the boundary locations of variables with "ALIGNALL(n)" in an LCF file. ALIGNALL(n) specifies the location of ELF sections. With the previous compiler, each variable was put into a separate section, so each section worked as an indication for a variable as a result.

With new compiler, variables may not be located at the specified location except the variable at the top of the ELF section. The inner section locations for each variable are determined by the variable data types. If you want to specify the boundary for a variable in this compiler, please use __attribute__ ((aligned(n))) in the source file.

2.4 Handling of locally unreferenced global data definitions

For the most part, the ARM3.0 compiler tries to place all data in pooled ELF sections. That is, the data is placed in common physical sections. Since the relative locations of data are known in pooled sections, access to the data is more efficient. The drawback of this is that the linker is unable to dead-strip unreferenced data since the linker can only strip the entire section.

To overcome this deficiency, the compiler now recognizes whether global data definitions are locally referenced, meaning that the global data is referenced within the file in which it is declared. If the data is not locally referenced, it is placed in its own individual ELF section. This allows the linker to make the final decision on dead-stripping it. This feature is only active in -ipa file mode. (See ARM_Compiler_Notes.txt.)

2.5 Unreferenced Static Symbol Dead-Stripping by the Compiler and its Prevention

#pragma force_func_export on|off|reset

This pragma will force a function to be viewed as referenced by the compiler's code dead-stripper in -ipa file mode. This is useful for files that contain only C static declared functions. Normally, the compiler would simply dead-strip these functions since they have internal linkage and no references. Pragma force_func_export will mark the functions as referenced, preventing dead-stripping in the compiler in -ipa file mode. The scope of #pragma force_func_export remains in effect for each function parsed after the pragma until it is reset or set again. For example:

```
#pragma force_func_export on
func1() {}
#pragma force_func_export off
func2() {}
```

force_func_export is enabled for func1() and disabled for func2() regardless of ipa mode settings.

The behavior of the ARM 3.0 compiler in -ipa file mode differs from the ARM 2.x compiler in deferred code generation mode. The behavior in -ipa file mode is to dead-strip internal linkage object than are not referenced. Care must be exercised to address cases where unreferenced objects may have internal linkage. Whereas previous compilers may have exported these objects to the linker, the ARM3.0 compiler will dead-strip the objects. In particular, this happens in the NITRO SDK in build/demos/fs/overlay-staticinit/src/top_menu.c and include/nitro/sinit.h.

There are several options to make sure that unreferenced objects are exported to the linker. For functions, use pragma force_func_export as described above. The __declspec (force_export) or __attribute__((force_export)) may be applied to individual objects to export them to the linker. This __declspec(or __attribute__) may be used on data or functions.

Examples:

// Somedata will not be dead-stripped in -ipa file mode

```
__declspec(force_export) static int Somedata[] = { 1,2 };
```

or

```
static int Somedata[] __attribute__((force_export)) = { 1,2 };
```

(ARM_Compiler_Notes.txt)

3 Build environment

3.1 Changes to Runtime Library and Include Paths

Due to changes in the runtime, the path to the runtime library is now

Runtime\Runtime_ARM\Runtime_NITRO\lib

Locations of some header files have also changed. The director below may have to be added to your project's paths in some cases:

ARM_EABI_Support\msl\MSL_Extras\MSL_ARM\Include

The makefiles in NITRO-SDK3.1 have already been updated to incorporate these changes.

3.2 Command-Line Flags Changes to Select Warning Levels

The command line flags to select warning levels have changed:

'-w most' enables the following warnings:

"illegal #pragmas",
"empty declarations",
"possible unwanted effects",
"unused arguments",
"unused variables",
"extra commas",
"pedantic error checking",
"hidden virtual functions",
"passing large arguments to unprototyped functions",

"use of expressions as statements without side effects",
"lossy conversions from pointers to integers",
"token not formed by ## operator",
"return without a value in non-void-returning function",
"inconsistent use of 'class' and 'struct'",
"incorrect capitalization used in #include \"...\"",
"incorrect capitalization used in #include <...>",

'-w all' enables the above warnings and the following:

"implicit arithmetic conversions; implies '-warn impl_float2int,impl_signedunsigned'",
"implicit integral to floating conversions",
"implicit floating to integral conversions",
"implicit signed/unsigned conversions",

'-w full' enables all of the above and the following:

"use of undefined macros in #if/#elif conditionals",
"'inline' functions not inlined",
"padding added between struct members",
"result of non-void-returning function not used",
"any conversions from pointers to integers",

(See ARM_Compiler_Notes.txt)

3.3 Smaller Code Size Libraries for Fixed Point Divide

Fixed point divide routine size was reduced by re-rolling a loop. Overall mathlib size has been reduced due to improved dead-stripping of unused math functions. Users may now link in "_sz.a" type libraries to get improvement in code size of divide routines for 32-bit signed and unsigned divide routine. But the smaller routines are slower than the unrolled loop. The non-"_sz.a" libraries are smaller also due to dead-stripping, but have the same performance as the previous libraries. (ARM_Compiler_Notes.txt)

The "_sz libraries" are those denoted with an “_sz.a” a the end of the library’s file name. The following 12 libraries have been added.

FP_fastI_v4t_LE_sz.a FP_fastI_v5t_LE_sz.a FP_fastI_xM_LE_sz.a
FP_fixedI_v4t_LE_sz.a FP_fixedI_v5t_LE_sz.a FP_fixedI_xM_LE_sz.a
FP_flush0_v4t_LE_sz.a FP_flush0_v5t_LE_sz.a FP_flush0_xM_LE_sz.a
FP_fullI_v4t_LE_sz.a FP_fullI_v5t_LE_sz.a FP_fullI_xM_LE_sz.a

You can use the _sz version libraries as follows:

With IDE projects

For each target, remove FP_fastI_v5t_LE.a and add FP_fastI_v5t_LE_sz.a at the overlay tab on the project window.

When using Makefile

Change the file

$(NITROSDK_ROOT)\build\buildtools\commondefs.cctype.CW

In this file, specify one of the _sz libraries for the CW_LIBFP value. Or, in the makefile for your target binary, specify _sz library for the make variable CW_LIBFP.

3.4 Removal of *strb.a libraries

The *strb.a’ libraries which were provided in CW for DS1.2 have been removed in CW for DS 2.0 and later.

4 New features in the compiler

4.1 Support for Java-style "override"/ "final" Error Checking

"final" and "override" __declspec()/__attribute__() specifiers can now be used for improved function override checking.

```
    struct A {
        virtual __declspec(final) void vf1();
        virtual void vf2();
    };
    struct B : A {
        void vf1();                         // ERROR
    };
    struct C : A {
        __declspec(override) void vf2();    // OK
        __declspec(final) void vf3();       // OK
    };
    struct D : A {
        __declspec(override) void vf();     // ERROR
    };
    struct E {
        __declspec(override) void vf();     // ERROR
    };
 =>
    Error   : the 'final' function 'A::vf1()' is overridden by 'B::vf1()
'
    Test.cpp line 7   };

    Error   : the 'override' function 'D::vf()' does not override
              any inherited functions
    Test.cpp line 14   };

    Error   : the 'override' function 'E::vf()' does not override
              any inherited functions
    Test.cpp line 17   };
```

(CW C-C++ Notes 4.x.txt)

4.2 Use the Pragma #define_section to Place const Qualified Data in a User-specified Section

User specified read-only data sections may be specified with
#pragma define_section.

For example:
#pragma define_section mysection, ".mydata", ".mybss", ".myrodata", abs32, RWX

Use this pragma to place const qualified data in a user-specified section. Otherwise, const data is placed in .rodata sections.

4.3 Improved "uninitialized variable" Warning

Added #pragma warn_possiblyuninitializedvar on | off | reset (default: off)

This option, when on, will force the compiler to test for a definition (assignment) of a variable on _all_ data flow paths to the use of that variable before emitting an uninitialized variable warning. The behavior, when off, is to test for a definition of a variable on _any_ data flow path to the use of a variable before emitting a warning.

```
static int test()
{
    int a;
    while (1) {
        a = sub(a);
        if(a)
            break;
    }
   return a;
}
```

Normally, no warning is generated for this case because a path from the definition of "a" (a = sub(a)) to the use of "a" (sub(a)) exists within the loop . With #pragma warn_possiblyuninitializedvar on, the compiler forces a check of _all_ paths, including the entry into the loop. Since the path into the entry of the loop contains a use of "a" without an initialization of "a", a warning is generated for the use of an uninitialized variable.

4.4 Disabling "permit dead-stripping of functions" and Disabling "Pool Strings" at the Same Time

The "Pool Strings" options in the C/C++ Language panel now control the pooling of constants. When "Permit dead-stripping of functions" is disabled and "Pool Strings" is disabled, each data object will occupy a unique ELF section. Formerly, "Permit dead-stripping of functions" when disabled would group all data objects into a single ELF section, regardless of "Pool Strings" settings.

In this implementation, only the string data objects are un-pooled if "Pool Strings" is disabled. The bulk of the data objects remains pooled. This leads to a slight performance reduction that can be regained if "Pool Strings" is enabled. By default, "Pool Strings" is disabled. So, ultimately, most of the data in a translation unit is pooled into a single ELF .data section. Each individual string resides in a unique ELF .data section when "Pool Strings" is off.

A benefit of this feature is that strings can be globally aligned in the linker control file with an align directive.

Example:

```
.align all 8
.data(*)
```

A caveat of this feature is that some unreferenced strings that possibly were not dead-stripped previously by the linker can now be dead-stripped because they live in unique ELF .data sections.
(ARM_Compiler_Notes.txt)

5 New features in the linker

5.1 Showing a Warning for Different structs with the Same Name in Different Files

This feature will detect when class/struct/union with the same struct/class/union names are defined differently in separate files.

Debugging must be enabled in the compilation of the files to be checked, that is, DWARF2 debugging information (.debug_info sections, etc.) must be present in the files being checked.

When the linker comes across 2 DWARF2 class/struct/union type symbols with the same name, it first checks whether one of these are a place-holder type (that is, a forward declaration having a struct size of 0). The place-holder type will be replaced by the other class/struct/union type. If both are place-holder types, then the newer one will be ignored.

If both symbols are not place-holder type, the linker will then check for the size of the struct of each symbol, and generate a warning if they are different. The linker will also ignore the newer one. Note this affects only the DWARF2 symbols.

The following warning is emitted if the the class/struct/union sizes are different:

Warning : The name <name> has been used for different structs/classes/unions.

(ARM_Linker_Notes.txt)

5.2 Using Duplicate Filenames in Different Overlays

Linker now allows duplicate filenames in different overlays.

The linker now allows the following example to link properly:

```
.sec1:ovl1 {
    file1.o (.text)
    file3.o (.text)
} > seg1

.sec2:ovl2 {
    file2.o (.text)
    file3.o (.text)
} > seg2
```

In this case, file3.o is a component of the overlays "ovl1" and "ovl2".

Note the new linker control file syntax that allows the overlay name to be specified explicitly following the section specifier. The overlay name is optional. It is only required when a file or files is shared among overlays. The overlay name must match the name of the overlay specified on the command line with the -overlay (-ol) flag.

The above syntax is equivalent to

```
.sec1: {
    GROUP(ovl1) (.text)
} > seg1

.sec2: {
    GROUP(ovl2) (.text)
} > seg2
```

Note that in the usage of GROUP(<name>), <name> refers to an overlay specified with -overlay, _not_ the overlay group name specified with -overlaygroup.

Library names may be used in place of object file names:

Example:

```
.sec1:ovl1 {
    file1.o (.text)
    lib1.a (.text)
} > seg1

.sec2:ovl2 {
```

```
    file2.o (.text)
    lib1.a (.text)
} > seg2
```

Normally, a warning is issued for duplicate file names on the command line. No warning is issued for duplicate files that exist as part of an overlay group.

Files may not be shared in overlays in different overlay groups.

This feature applies only to the command line linker, as the IDE does not have the ability to duplicate a file in overlay groups. The CodeWarrior debugger may not work correctly if the binary file was built with this feature.

(ARM_Linker_Notes.txt)

5.3 Use of Duplicate Symbols and Control of Symbol Resolution

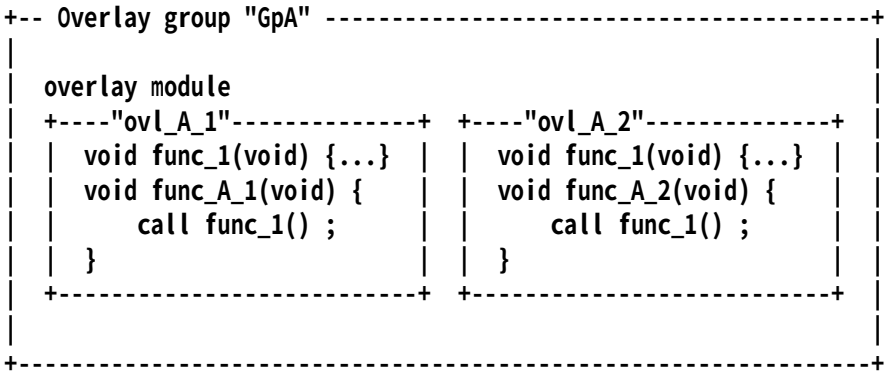
With the linker in CW for DS1.2, duplicate symbol definitions were only allowed in different overlays of the same overlay group.
The linker in CW for DS2.0 allows duplicate symbols to be defined in different overlay groups. References to multiply defined symbols can be resolved with the SEARCH_SYMBOL directive in the linker control file (LCF).
This feature applies only to the command line linker. A build operation using this feature from the IDE is not supported. The CodeWarrior debugger may not work correctly if the binary file was built with this feature.

Example Cases:

```
Case 1:
+--- MAIN ---------+
|                  |
|  call func_x()   |
|  call func_y()   |
|                  |
+------------------+
```

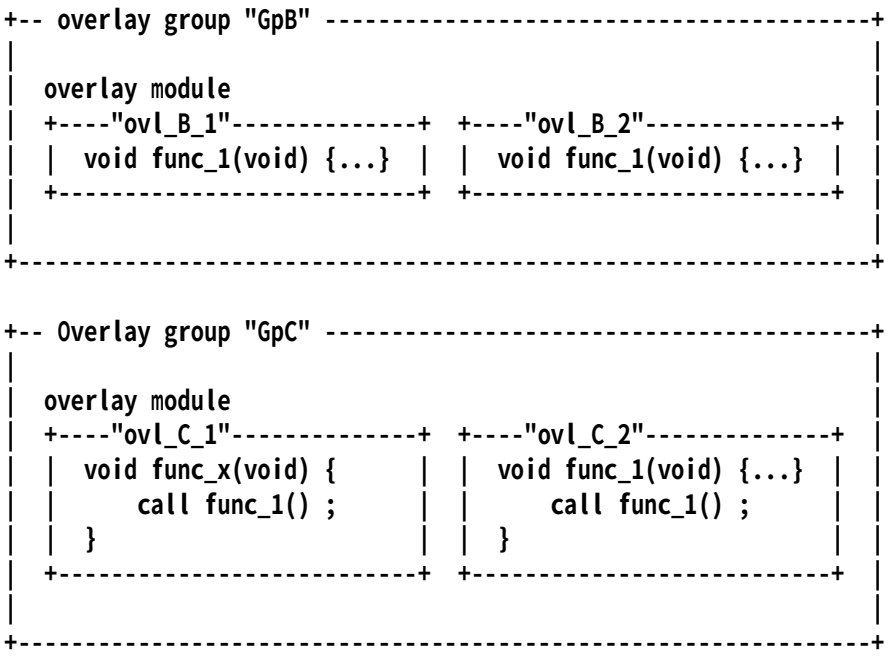
The MAIN wishes to call func_x() which resides in the overlay module ovl_C_1, an d the MAIN wishes to call func_y() which resides in the overlay module ovl_C_2.

Case 2:



The overlay module ovl_A_1 wishes to call func1() which resides in the overlay module ovl_A_1 itself, and the overlay module ovl_A_2 wishes to call func1() which resides in the overlay module ovl_A_2 itself.

Case 3:



The overlay module ovl_C_1 wishes to call func_1() which resides in overlay module ovl_B_1, and the overlay module ovl_C_2 wishes to call func_1() which resides in overlay module ovl_B_2.

To solve these cases, symbol resolution rules must be specified. The new LCF directive SEARCH_SYMBOLS has been introduced to specify these rules. This directive can be used in SECTIONS section in LCF.

Syntax

SEARCH_SYMBOL {search name} [, {search name} ..] ;

Parameter

{search name} : overlay group name, or, overlay module name

Notes

The linker searches for the symbol start from the calling module itself. Next, it searches for the symbol from the overlay groups or overlay modules specified by the SEARCH_SYMBOL directive. If the symbol could not be found, the linker emits an error.

Examples:

```
SECTIONS
{
    .main:
    {
        SEARCH_SYMBOL ovl_C_1, ovl_C_2 ;
    } > main
    .main.bss:
    {
        SEARCH_SYMBOL ovl_C_1, ovl_C_2 ;
    } >> main
    .ovl_A_1:
    {
        SEARCH_SYMBOL ovl_A_1 ;
    } > ovl_A_1
    .ovl_A_1.bss:
    {
        SEARCH_SYMBOL ovl_A_1 ;
    } >> ovl_A_1
    .ovl_A_2:
    {
        SEARCH_SYMBOL ovl_A_2 ;
    } > ovl_A_2
    .ovl_A_2.bss:
    {
        SEARCH_SYMBOL ovl_A_2 ;
```

```
    } >> ovl_A_2
    .ovl_C_1:
    {
        SEARCH_SYMBOL ovl_B_1 ;
    } > ovl_C_1
    .ovl_C_1.bss:
    {
        SEARCH_SYMBOL ovl_B_1 ;
    } >> ovl_C_1
    .ovl_C_2:
    {
        SEARCH_SYMBOL ovl_B_2 ;
    } > ovl_C_2
    .ovl_C_2.bss:
    {
        SEARCH_SYMBOL ovl_B_2 ;
    } >> ovl_C_2
}
```

A more detailed explanation for this feature is available at ARM_Linker_Notes.txt.

6 New Features in the Inline Assembler

6.1 Label Operands in ldr and lda Instructions

The compiler's inline assembler now accepts label operands in ldr and lda instructions.

Example

```
        asm {
                ldr r0,=L1
                lda r1,L2
L1:
L2:
        }
```

where L1 and L2 are legal inline assembly labels. Labels must be defined in the function in which they are referenced.
(ARM_Compiler_Notes.txt)

6.2 Optional Trailing Colon for Label

Inline assembly labels may now have an optional trailing colon.

Sample 1

```
    asm {
    label1
    b @label1
}
```

Sample 2

```
    asm {
label1:
    b @label1
}
```

The two samples above are identical. The label may begin anywhere in the line. The lines containing a label may or may not contain additional opcodes.
(ARM_Compiler_Notes.txt)

6.3 .ALIGN Directive

The compiler's inline assembler now accepts an .align directive
The usage is:

```
.align <arg>
```

where <arg> is a literal constant - expressions are not allowed. <arg> must be a power of 2 constant greater than 0. The maximum align argument is 8192.

.align aligns the next occurring instruction or directive to the argument value. Multiple .align directives are allowed in an inline assembly block.

Normally the function where the .align directive resides will be aligned to the maximum of all the .align directives in the function. This is true if dead-stripping of functions is enabled.

If dead-stripping is disabled in the compiler (target settings panel "Permit dead-stripping of functions" is unchecked), individual functions may not be necessarily aligned to the maximum of all .align directives in the function. By disabling dead-stripping, the compiler essentially combines functions in order to pool constants and gain more efficient access of constant addresses of global and static variables. All the combined functions will reside in the same object code section and the object code section is aligned to the maximum of the .align directives. Do not rely on function alignment if dead-stripping is disabled. The .align directives will be accurate, however.
(ARM_Compiler_Notes.txt)

7 Notice

7.1 A Case in which Auto Inlining Works Even if -inline noauto is Specified

When compiling, if the all of these three conditions exist, functions will be inlined automatically, even though the auto inline was set to OFF.

- "optimize for size" is selected
- "dont inline" is not specified.
- If inlined, the code size will be smaller then non inlined (when the function is called only once)

To prevent this auto inlining, use __attribute__((never_inline)) for the prototype of the function which you don't want it to be inlined.

* __attribute__((never_inline)) can be used both CW for DS1.2 and CW for DS2.0.

7.2 Detailed Control of Inlining

With the compiler in CW for DS2.0, there are other ways to control inlining in addition to __attribute__((never_inline)).

__attribute__((noinlining))
__declspec(noinlining)

This function declaration specifier will prevent any inlining in the body of a function. It does not prevent the inlining of the function itself.

```
inline void f() {}
inline void g() __attribute__((noinline)) {}

void foo()
{
   f();   //     will be inlined
   g();   //     will not be inlined
```

```
}

void bar() __attribute__((noinlining))
{
   f();   //     will not be inlined
   g();   //     will not be inlined
}

void foobar() __attribute__((noinline, noinlining))
{
   //     this function will not be inlined and nothing
   //     in this function will be inlined
}
```

Functions that are defined in a "#pragma dont_inline on" section of the source text will have an implicit "noinlining" specifier, so a function call in these functions will never be inlined, even if they are instantiated/generated at a point where #pragma dont_inline is off.
(CW C-C++ Notes 4.x.txt)

7.3 Read the version of the compiler from source files

You can use the predefined symbol __MWERKS__ to examine the version of the running compiler from a source file.

The following are defined values for this symbol for each version of CW for DS:

CodeWarrior for NINTENDO DS 1.2 SP3	0x3205
CodeWarrior for NINTENDO DS 2.0	0x4020

This constant value is not the exact compiler version. Instead, it indicates the version of the compilers Front End (language processor.)
A similar predefined symbol, __CWCC__ is described in the Build Tools Reference, but it is actually not defined in the current compiler used in CW for DS2.0.

7.4 CW C-C++ Notes 4.x.txt

The compiler's Front End has been updated since CW for DS1.2. A detailed explanation of the new Front End is available at

Release_Notes\ARM\Compiler_ Release_ Notes\ CW C-C++ Notes 4.x.txt

Note that the following features described in the above mentioned notes are not supported by the compiler in CW for DS 2.0.

__attribute__((packed))

Although the __attribute__((packed)) is explained under the topic "Improved support for __attribute__((aligned(x))) / __attribute__((packed))", the compiler in CW for DS2.0 does not support this feature.

__attribute__((dllimport)) and __attribute__((dllexport))

Although the __attribute__((dllimport)) and the __attribute__((dllexport)) are introduced under the topic "New Attributes", the compiler in CW for DS2.0 does not support them.