

Technical Support

Overview
Search
Contact
Assistance Request
Feedback

On-Line Manuals

Product Manuals
Document Conventions

Compiler User Guide

- Preface
- Overview of the Compiler
- Getting Started with the Compiler
- Compiler Features
- Compiler Coding Practices
- Compiler Diagnostic Messages
- Using the Inline and Embedded Assemblers of the ARM
- Compiler Command-line Options
- Language Extensions
- Compiler-specific Features
 - Keywords and operators
 - __align
 - __ALIGNOF__
 - __alignof__
 - __asm
 - __forceinline
 - __global_reg
 - __inline
 - __int64
 - __INTADDR__
 - __irq
 - __packed
 - __pure
 - __smc
 - __softfp
 - __svc
 - __svc_indirect
 - __svc_indirect_r7
 - __value_in_regs
 - __weak
 - __writeonly
 - __declspec attributes
 - __declspec(noinline)
 - __declspec(noreturn)
 - __declspec(nothrow)
 - __declspec(notshared)
 - __declspec(thread)
 - Function attributes
 - __attribute__((alias)) function attribute
 - __attribute__((always_inline)) function attribute
 - __attribute__((const)) function attribute
 - __attribute__((deprecated)) function attribute
 - __attribute__((destructor(priority))) function attribute
 - __attribute__((format)) function attribute
 - __attribute__((format_arg(string-index))) function attribute
 - __attribute__((malloc)) function attribute
 - __attribute__((noinline)) function attribute
 - __attribute__((no_instrument_function)) function attribute
 - __attribute__((nomerge)) function attribute
 - __attribute__((nonnull)) function attribute
 - __attribute__((noreturn)) function attribute
 - __attribute__((notacall)) function attribute
 - __attribute__((nothrow)) function attribute
 - __attribute__((pcs("calling_convention"))) funcio
 - __attribute__((pure)) function attribute
 - __attribute__((section("name"))) function attribut
 - __attribute__((sentinel)) function attribute
 - __attribute__((unused)) function attribute
 - __attribute__((used)) function attribute
 - __attribute__((visibility("visibility_type"))) fun
 - __attribute__((warn_unused_result))
 - __attribute__((weak)) function attribute
 - __attribute__((weakref("target"))) function attrib
 - Type attributes
 - __attribute__((bitband)) type attribute
 - __attribute__((aligned)) type attribute
 - __attribute__((packed)) type attribute
 - Variable attributes
 - __attribute__((alias)) variable attribute
 - __attribute__((at(address))) variable attribute
 - __attribute__((aligned)) variable attribute
 - __attribute__((deprecated)) variable attribute
 - __attribute__((noinline)) constant variable attrib
 - __attribute__((packed)) variable attribute
 - __attribute__((section("name"))) variable attribut
 - __attribute__((unused)) variable attribute
 - __attribute__((used)) variable attribute
 - __attribute__((visibility("visibility_type"))) var
 - __attribute__((weak)) variable attribute
 - __attribute__((weakref("target"))) variable attrib
 - __attribute__((zero_init)) variable attribute
 - Pragmas
 - #pragma anon_unions, #pragma no_anon_unions
 - #pragma arm
 - #pragma arm section (section_type, list)
 - #pragma diag_default tag[,tag,...]
 - #pragma diag_error tag[,tag,...]
 - #pragma diag_remark tag[,tag,...]
 - #pragma diag_suppress tag[,tag,...]
 - #pragma diag_warning tag[, tag, ...]
 - #pragma exceptions_unwind, #pragma no_exceptions_u
 - #pragma GCC system_header
 - #pragma hdrstop
 - #pragma import symbol_name
 - #pragma import(__use_full_stdio)
 - #pragma import(__use_smaller_memcpy)
 - #pragma inline, #pragma no_inline
 - #pragma no_pch
 - #pragma Onum
 - #pragma once
 - #pragma Ospace
 - #pragma Otime
 - #pragma pack(n)
 - #pragma pop
 - #pragma push
 - #pragma softfp_linkage, #pragma no_softfp_linkage
 - #pragma thumb
 - #pragma unroll {n}
 - #pragma unroll_completely
 - #pragma weak symbol, #pragma weak symbol1 = symbol
 - Instruction intrinsics
 - __breakpoint intrinsic
 - __cdp intrinsic
 - __crex intrinsic
 - __clz intrinsic
 - __current_pc intrinsic
 - __current_sp intrinsic
 - __disable_fiq intrinsic
 - __disable_irq intrinsic
 - __dmb intrinsic
 - __dsb intrinsic
 - __enable_fiq intrinsic
 - __enable_irq intrinsic
 - __fabs intrinsic
 - __fabef intrinsic
 - __force_loads intrinsic
 - __force_stores intrinsic
 - __isb intrinsic
 - __ldrex intrinsic
 - __ldrexh intrinsic
 - __ldrt intrinsic
 - __memory_changed intrinsic
 - __nop intrinsic
 - __pld intrinsic
 - __pli intrinsic
 - __qadd intrinsic
 - __qdbl intrinsic
 - __qsub intrinsic
 - __rbit intrinsic
 - __rev intrinsic
 - __return_address intrinsic
 - __ror intrinsic
 - __schedule_barrier intrinsic
 - __semihost intrinsic
 - __sev intrinsic
 - __sgt intrinsic
 - __sgtf intrinsic
 - __ssat intrinsic
 - __strex intrinsic
 - __stret intrinsic
 - __swp intrinsic
 - __usat intrinsic
 - __wfe intrinsic
 - __wfi intrinsic
 - __yield intrinsic
 - ARMv6 SIMD intrinsics
 - ETSI basic operations
 - C5x intrinsics
 - VFP status intrinsic
 - __vfp_status intrinsic
 - Fused Multiply Add (FMA) intrinsics
 - Named register variables
 - GNU built-in functions
 - Predefined macros
 - Built-in function name variables
- C and C++ Implementation Details
 - What is Semihosting?
 - Via File Syntax
 - Summary Table of GNU Language Extensions
 - Standard C Implementation Definition
 - Standard C++ Implementation Definition
 - C and C++ Compiler Implementation Limits

Home / Compiler User Guide

__nop intrinsic

Home » Compiler-specific Features » __nop intrinsic

9.123 __nop intrinsic

This intrinsic inserts a `NOP` instruction or an equivalent code sequence into the instruction stream.

Syntax

```
void __nop(void)
```

Usage

The compiler does not optimize away the `NOP` instructions, except for normal unreachable code elimination. One `NOP` instruction is generated for each `__nop` intrinsic in the source. ARMv6 and previous architectures do not have a `NOP` instruction, so the compiler generates a `MOV r0,r0` instruction instead.

In addition, `__nop` creates a special sequence point that prevents operations with side effects from moving past it under all circumstances. Normal sequence points allow operations with side effects past if they do not affect program behavior. Operations without side effects are not restricted by the intrinsic, and the compiler can move them past the sequence point. The `__schedule_barrier` intrinsic also creates this special sequence point, and at optimization level `-O0` emits two `NOP` instructions. These instructions are removed at other optimization levels.

Section 5.1.2.3 of the C standard defines operations with side effects as those that change the state of the execution environment. These operations:

- Access volatile objects.
- Modify a memory location.
- Modify a file.
- Call a function that does any of the above.

Examples

In the following example, the compiler ensures that the read from the volatile variable `x` is enclosed between two `NOP` instructions.

```
volatile int x;  
int z;  
int read_variable(int y)  
{  
    int i;  
    int a = 0;  
    __nop();  
    a = x;  
    __nop();  
    return z + y;  
}
```

If the `__nop` intrinsics are removed, and the compilation is performed at `-O3 -Otime` for `--cpu=Cortex-M3`, for example, then the compiler can schedule the read of the non-volatile variable `z` to be before the read of variable `x`.

In the following example, the compiler ensures that the write to variable `z` is enclosed between two `NOP` instructions.

```
int x;  
int z;  
int write_variable(int y)  
{  
    int i;  
    for (i = 0; i < 10; i++)  
    {  
        __nop();  
        z = y;  
        __nop();  
        x += y;  
    }  
    return z;  
}
```

In this case, if the `__nop` intrinsics are removed, then with `-O3 -Otime --cpu=Cortex-A8`, the compiler can fold away the loop.

In the following example, because `pure_func` has no side effects, the compiler can move the call to it outside of the loop. Still, the compiler ensures that the call to `func` is enclosed between two `NOP` instructions.

```
int func(int x);  
int pure_func(int x) __pure;  
int read(int x)  
{  
    int i;  
    int a=0;  
    for (i=0; i<10; i++)  
    {  
        __nop();  
        a += pure_func(x) + func(x);  
        __nop();  
    }  
    return a;  
}
```

Note

- You can use the `__schedule_barrier` intrinsic to insert a scheduling barrier without generating a `NOP` instruction.
- In the examples above, the compiler would treat `__schedule_barrier` in the same way as `__nop`.

Related reference

9.13 `__pure`
9.133 `__schedule_barrier` intrinsic
3.4 *Generic intrinsics*
9.135 `__sev` intrinsic
9.144 `__wfe` intrinsic
9.145 `__wfi` intrinsic
9.146 `__yield` intrinsic

Related information

[NOP](#)

Products

Development Tools
Arm
C166
C51
C251
µVision IDE and Debugger

Hardware & Collateral

MDK-Arm
ULINK Debug Adaptors
Evaluation Boards
Product Brochures
Device Database
Distributors

Downloads

MDK-Arm
C51
C166
C251
File downloads

Support

Knowledgebase
Discussion Forum
Product Manuals
Application Notes

Contact

Distributors
Request a Quote
Sales Contacts