

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

The compiler as an optimizing compiler

Compiler optimization for code size versus speed

Compiler optimization levels and the debug view

Selecting the target processor at compile time

Enabling FPU for bare-metal

Optimization of loop termination in C code

Loop unrolling in C code

Compiler optimization and the volatile keyword

Code metrics

Code metrics for measurement of code size and data

Stack use in C and C++

Benefits of reducing debug information

Privacy Policy Update

Important information

Arm's Privacy Policy has been updated. By continuing to use our site, you consent to Arm's Privacy Policy. Please review our [Privacy Policy](#) to learn more about our collection, use and transfers of your data.

Accept and hide this message

This site uses cookies to store information on your computer. By continuing to use our site, you consent to our [cookies](#).

Don't show this message again

Change Settings

[Home](#) / [Compiler User Guide](#)

Compiler optimization and the volatile keyword

[Home](#) » [Compiler Coding Practices](#) » Compiler optimization and the volatile keyword

4.8 Compiler optimization and the volatile keyword

Higher optimization levels can reveal problems in some programs that are not apparent at lower optimization levels, for example, missing `volatile` qualifiers.

This can manifest itself in a number of ways. Code might become stuck in a loop while polling hardware, multi-threaded code might exhibit strange behavior, or optimization might result in the removal of code that implements deliberate timing delays. In such cases, it is possible that some variables are required to be declared as `volatile`.

The declaration of a variable as `volatile` tells the compiler that the variable can be modified at any time externally to the implementation, for example, by the operating system, by another thread of execution such as an interrupt routine or signal handler, or by hardware. Because the value of a `volatile`-qualified variable can change at any time, the actual variable in memory must always be accessed whenever the variable is referenced in code. This means the compiler cannot perform optimizations on the variable, for example, caching its value in a register to avoid memory accesses. Similarly, when used in the context of implementing a sleep or timer delay, declaring a variable as `volatile` tells the compiler that a specific type of behavior is intended, and that such code must not be optimized in such a way that it removes the intended functionality.

In contrast, when a variable is not declared as `volatile`, the compiler can assume its value cannot be modified in unexpected ways. Therefore, the compiler can perform optimizations on the variable.

The use of the `volatile` keyword is illustrated in the two sample routines of the following table. Both of these routines loop reading a buffer until a status flag `buffer_full` is set to true. The state of `buffer_full` can change asynchronously with program flow.

The two versions of the routine differ only in the way that `buffer_full` is declared. The first routine version is incorrect. Notice that the variable `buffer_full` is not qualified as `volatile` in this version. In contrast, the second version of the routine shows the same loop where `buffer_full` is correctly qualified as `volatile`.

Table 4-5 C code for nonvolatile and volatile buffer loops

Nonvolatile version of buffer loop	Volatile version of buffer loop
<pre>int buffer_full; int read_stream(void) { int count = 0; while (!buffer_full) { count++; } return count; }</pre>	<pre>volatile int buffer_full; int read_stream(void) { int count = 0; while (!buffer_full) { count++; } return count; }</pre>

The following table shows the corresponding disassembly of the machine code produced by the compiler for each of the examples above, where the C code for each implementation has been compiled using the option `-O2`.

Table 4-6 Disassembly for nonvolatile and volatile buffer loop

Nonvolatile version of buffer loop	Volatile version of buffer loop
<pre>read_stream PROC LDR r1, [L1.28] MOV r0, #0 LDR r1, [r1, #0] [L1.12] CMP r1, #0 ADDEQ r0, r0, #1 BEQ [L1.12] ; infinite loop</pre>	<pre>read_stream PROC LDR r1, [L1.28] MOV r0, #0 [L1.8] LDR r2, [r1, #0]; ; buffer_full CMP r2, #0 ADDEQ r0, r0, #1 BEQ [L1.8]</pre>

PDFmyURL easily turns web pages and even entire websites into PDF!

PDFmyURL

Inline functions and removal of unused out-of-line
Automatic function inlining and multfile compilat
Restriction on overriding compiler decisions about
Compiler modes and inline functions
Inline functions in C++ and C90 mode
Inline functions in C99 mode
Inline functions and debugging
Types of data alignment
Advantages of natural data alignment
Compiler storage of data objects by natural byte a
Relevance of natural data alignment at compile tim
Unaligned data access in C and C++ code
The __packed qualifier and unaligned data access i
Unaligned fields in structures
Performance penalty associated with marking whole
Unaligned pointers in C and C++ code
Unaligned Load Register (LDR) instructions generat
Comparisons of an unpacked struct, a __packed stru
Compiler support for floating-point arithmetic
Default selection of hardware or software floating
Example of hardware and software support differenc
Vector Floating-Point (VFP) architectures
Limitations on hardware handling of floating-point
Implementation of Vector Floating-Point (VFP) supp
Compiler and library support for half-precision fl
Half-precision floating-point number format
Compiler support for floating-point computations a
Types of floating-point linkage
Compiler options for floating-point linkage and co
Floating-point linkage and computational requireme
Processors and their implicit Floating-Point Units
Integer division-by-zero errors in C code
Software floating-point division-by-zero errors in
About trapping software floating-point division-by
Identification of software floating-point division
Software floating-point division-by-zero debugging
New language features of C99
New library features of C99
// comments in C99 and C90
Compound literals in C99
Designated initializers in C99

```

    BX      lr
    ENDP
|L1.28|
    DCD     ||.data||
    AREA   ||.data||, DATA, ALIGN=2
buffer_full
    DCD     0x00000000
```

```

    BX      lr
    ENDP
|L1.28|
    DCD     ||.data||
    AREA   ||.data||, DATA, ALIGN=2
buffer_full
    DCD     0x00000000
```

In the disassembly of the nonvolatile version of the buffer loop in the above table, the statement `LDR r0, [r0, #0]` loads the value of `buffer_full` into register `r0` outside the loop labeled `|L1.12|`. Because `buffer_full` is not declared as `volatile`, the compiler assumes that its value cannot be modified outside the program. Having already read the value of `buffer_full` into `r0`, the compiler omits reloading the variable when optimizations are enabled, because its value cannot change. The result is the infinite loop labeled `|L1.12|`.















In contrast, in the disassembly of the volatile version of the buffer loop, the compiler assumes the value of `buffer_full` can change outside the program and performs no optimizations. Consequently, the value of `buffer_full` is loaded into register `r0` inside the loop labeled `|L1.8|`. As a result, the loop `|L1.8|` is implemented correctly in assembly code.

To avoid optimization problems caused by changes to program state external to the implementation, you must declare variables as `volatile` whenever their values can change unexpectedly in ways unknown to the implementation.

In practice, you must declare a variable as `volatile` whenever you are:





- Accessing memory-mapped peripherals.
- Sharing global variables between multiple threads.
- Accessing global variables in an interrupt routine or signal handler.













The compiler does not optimize the variables you have declared as `volatile`.

-  Hexadecimal floating-point numbers in C99
-  Flexible array members in C99
-  __func__ predefined identifier in C99
-  inline functions in C99
-  long long data type in C99 and C90
-  Macros with a variable number of arguments in C99
-  Mixed declarations and statements in C99
-  New block scopes for selection and iteration state
-  _Pragma preprocessing operator in C99
-  Restricted pointers in C99
-  Additional library functions in C99
-  Complex numbers in C99
-  Boolean type and in C99
-  Extended integer types and functions in



floating-point environment access in C99

-  snprintf family of functions in C99
-  type-generic math macros in C99
-  wide character I/O functions in C99
-  How to prevent uninitialized data from being initi

-  Compiler Diagnostic Messages
-  Using the Inline and Embedded Assemblers of the AR
-  Compiler Command-line Options
-  Language Extensions
-  Compiler-specific Features
-  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

Products

Development Tools
Arm
C166
C51
C251
µVision IDE and Debugger

Hardware & Collateral
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

