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

```
int buffer_full;
int read_stream(void)
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
```

Volatile version of buffer loop

```
volatile int buffer_full;
int read_stream(void)
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
```

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

```
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
```

Volatile version of buffer loop

```
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|
```

```
    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.