Other possible causes are listed here.

16. **When should I use volatile?**

The volatile qualifier originates with the "as-if" rule in C:

> In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

The critical part here is, "An actual implementation need not evaluate part of an expression if it can deduce that its value is not used". For example,

```
int i;

PORTA = 1;

for (i = 0; i < 10; i++)
    ;

PORTA = 2;
```

Your expectation might be that 1 would be written to PORTA, then a delay, then 2 would be written to PORTA. But the compiler is free to just do "PORTA = 2", or even to generate no code at all. This is because a compiler can look at the code and realize there are "no needed side effects" from assigning 1 to PORTA or from the loop.

But PORTA works as expected because it is declared as volatile in the compiler's header files; every operation on PORTA must be carried out exactly as stated by the programmer:

> An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine

"volatile" tells the compiler to do exactly what the programmer said to do. To correct the code fragment above, "i" must also be volatile to ensure the delay loop is executed exactly as stated.

volatile is also relevant in interrupt handlers in cases like this:

```
int interrupt_flag;

int main(void) {
    interrupt_flag = 0;

    while (interrupt_flag == 0)    /* wait for interrupt */
        ;

    /* do stuff */
}
```

The compiler can look at that code and realize that interrupt_flag is 0, and will always remain 0, so there is no need for it to generate any code at all for main(), even though you might have an interrupt handler somewhere that sets interrupt_flag to 1. The solution is to make interrupt_flag volatile, so that the compiler understands that it "may be modified in ways unknown to the implementation."

Note that volatile does *not* provide atomic access. In the previous example, it will take multiple CPU instructions to compare interrupt_flag against 0 on an eight-bit processor. The volatile keyword does not prevent an interrupt from occurring in the middle of this sequence of instructions, which means interrupt_flag may take on an unexpected value with a properly timed interrupt. Atomic access requires additional protection, such as disabling interrupts around the comparison.

17. **Should I just make everything volatile to be safe?**

No. With a modern C compiler, you have to forget about the idea that the compiler is going to blindly follow your instructions step by step. You have to think in terms of what side effects your program generates.

In an embedded program, side effects are almost exclusively caused by reading or writing the processor's special function registers. If you're blinking an LED, sending data over a UART, or writing to internal EEPROM, you are accessing a special register that is declared as volatile, so all of these things are guaranteed to happen. But other concepts, like assigning a value to a variable, or the passage of time caused by a delay loop, are not side effects. For example:

```
int main(void)
{
    int a, b;

    a = 1;
    a *= 2;
    b = 3;

    PORTA = a + b;
}
```

The only side effect in this program is writing 5 to PORTA. Since PORTA is volatile, we are guaranteed that will happen. What we aren't guaranteed is that some memory location called "a" will have the value 1 written to it, or that the processor's multiply instruction will be used in calculating the value to be written to PORTA. If the compiler is smart enough to replace the above with

```
int main(void)
{
    PORTA = 5;
}
```

we've achieved the same result with less code space and faster execution time.

There is one situation where you'd want to declare "a" and "b" volatile in the above code: when you're debugging. If your compiler is smart enough to optimize all of the assignments away, you won't be able to step through the assignments in the debugger to make sure they're working the way you want. Using the volatile keyword temporarily during debugging can work around this, as can choosing a lower optimization level during debugging.