Number:	P2327R0
Title:	De-deprecating volatile compound operations
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG14, SG1, SG22, EWG, WG21
Date:	2021-04-15
Authors:	Paul M. Bendixen <paulbendixen@gmail.com>
Contributors:	Jens Maurer
	Arthur O'Dwyer
	Ben Saks
Email:	paulbendixen@gmail.com
Reply to:	Paul M. Bendixen

Revision history

1 Introduction

The C++ 20 standard deprecated many functionalities of the volatile keyword. This was due to P1152[Bastien, 2019]. The reasoning is given in the R0 version of the paper[Bastien, 2018].

The deprecation was not received too well in the embedded world as volatile is commonly used for communicating with peripheral devices in microcontrollers[Ooijen, 2020].

The purpose of this paper is to show what parts of the deprecation that are critical to the embedded domain, and hopefully find a solution that satisfies the embedded community without sacrificing the work of P1152.

2 Problem statement

2.1 Background

One of the great advantages of C++ is its closeness to the machine on which it operates. This enables C++ to be used in very constrained devices such as microcontrollers without any operating system. These systems often operate by manipulating memory mapped registers often only touching a single bit.

While there are multiple ways to manipulate single bits in a memory location the most idiomatic way is something along the lines of the following:

```
// In vendor supplied hardware abstraction layer
struct ADC {
        volatile uint8_t CTRL;
        volatile uint8_t VALUE;
        ...
};
#define ADC_CTRL_ENABLE (1 << 3)


// in user code
ADC1->CTRL |= ADC_CTRL_ENABLE; // Start the ADC
...
```

```
ADC1->CTRL &= ~ADC_CTRL_ENABLE; // Stop the ADC
```

This is further amplified by the fact that vendors supply macros or inline functions for setting or clearing bits utilizing this idiom, or may use it in provided code generators, as shown in the following example from a library for the Energy Micro (now Silicon Labs) series of ARM microcontrollers:

```
// Copyright 2012 Energy Micro EMLib from em_i2c.h
__STATIC_INLINE void I2C_IntDisable(I2C_TypeDef *i2c, uint32_t flags)
{
  i2c->IEN &= ~(flags);
}
```

It is clear from the previous example that this idiomatic usage of compound operations clashes with the deprecation of compound operations on volatile. The vendors of these libraries are almost always the chip vendors themselves and they will, based on experience from adapting C++ toolchains, not be likely to update their header files. The argument against compound operations on volatile variables is that it leads the programmer to believe that the operation itself becomes compounded and therefore atomic. While this is possible (https://godbolt.org/z/q5bn5K) it depends on the platform and the compiler.

Now since there on some platforms can be a read, modify, write cycle it is possible that an interrupt would happen in between the read and the write, however most embedded code that uses this idiom needs to take care of this by being right by construction i.e. not bit-twiddeling variables that are used in the interrupt service routines (ISRs). The problem being that there is currently no way to do this in a guaranteed atomic manner (see also [Craig, 2020, section 3.4.1]).

2.2 Scope

So what is the impact of deprecating compound operations on volatile? A search of some of the more widely used embedded libraries show that while its use is not massive, it is in some way in almost all hardware libraries for embedded systems available today.

The numbers below are done by using UNIX grep[1] to find usages of either |= or &= on variables with a code style usually associated with volatile members in the code bases of the respective libraries. While this method isn't foolproof it does give an indicator of the problem.

Library	compound operations
Silabs Gecko SDK	293
AVR libc 2.0	205
Raspberry pi Pico SDK	13

Table 1: Occurrences of compound operations on variables typically associated with volatile

This illustrates the major point, not only is there a lot of code already written out there working as expected but the usage of this idiom also prevents adopters of C++ to use the C libraries provided with their toolchains.

Furthermore, one of the most common ways to gradually change from C to C++ is by keeping the same code and compile it with a C++ compiler. (See e.g. [von Tessin, 2019, from 27:46]). This procedure involves modifying the code to be correct in C++ and would possibly involve moving compound volatile statements to non-compound statements. However the prevalence of this idiom

[1]The search was done using the command
`grep -re "[A-Z_0-9]\+\(\[.*\]\)*[ ]\+[&|]=" -include \*.h .`

in C headers would prevent this low cost approach and would require a larger up front investment creating new headers to replace the vendor supplied ones.

The idea that businesses will be willing to spend the effort to update existing codebases that become defunct because of this changes seems unlikely, rather it seems likely that they will mandate using no newer versions.

2.3 Error potential

One of the arguments for deprecation in [Bastien, 2018] is the potential for error for programmers unaccustomed to using volatile. However removal of the compound operators will also risk the introduction of errors such as in the following.

```
UART1->UCSR0B |= (1<<UCSZ01); // (1)
UART1->UCSR0B = UART1->UCSR0B | (1<<UCSZ01); // (2)
UART2->UCSR0B = UART1->UCSR0B | (1<<UCSZ01); // (3)
```

The code in (1) is the original code, setting a bit in a mapped register. In (2) the code is transformed to the style that is suggested as a replacement. (3) describes a possible error scenario, where the device is changed to another, but due to the code duplication of the updated style, an error has slipped in and the value of the old device is read.

An error such as the above will not necessarily be caught in code review, and will possibly not even be found in immediate testing if UART1 happens to have the correct setting during testing, such code is also notoriously hard to unittest. As such the deprecation will trade errors where volatile is erroneously used to express atomicity for hard to discover and hard to detect errors due to duplication.

2.4 Didn't we just go over this?

While the proposal to deprecate was heard in committee meetings, the main focus was on problems arising with multi-threaded code (SG1) and with EWG, neither of these groups can be expected to be familiar with the inner workings on microcontrollers.

3 Possible solutions

3.1 The simple

The simplest possible change that could possibly work would be to remove the text added to paragraph [expr.ass] point 6 as this would allow compound statements on volatile variables.

> The behavior of an expression of the form E1 op= E2 is equivalent to E1 = E1 op E2 except that E1 is evaluated only once. ~~Such expressions are deprecated if E1 has volatile-qualified type; see [depr.volatile.type].~~ For += and -=, E1 shall either have arithmetic type or be a pointer to a possibly cv-qualified completely-defined object type. In all other cases, E1 shall have arithmetic type.

Since this brings in non-simple assignments to volatile-qualified operands, the previous paragraph should be modified:

> ~~A simple~~An assignment whose left operand is of a volatile-qualified type is deprecated ([depr.volatile.type]) unless the (possibly parenthesized) assignment is a discarded-value expression or an unevaluated operand.

The examples in [depr.volatile.type] should be updated

```
brachiosaur += neck;                    //- deprecated
brachiosaur += neck;                    //+ OK
```

3.2 The compromise

As the compound operations are mainly used to flip bits, a compromise could be to only de-deprecate the use of binary compound operations (`|=` `&=` `^=` and possibly `>>=` `<<=`) as these are the ones that are useful for this purpose. In the examination of the libraries `+=` and `-=` did not occur, but they might in commercial / closed source libraries.

This would require new wording to be put in.

Built-in operators [over.built] point 25 would be affected in that only half the points would be affected.

4 Impact

The changes proposed by this paper would affect the current proposal P2139[Meredith, 2020].

5 Thanks

Thanks to Jens Maurer for the suggestion on the compromise solution.
Thanks to Wouter Van Ooijen for starting this discussion.
Thanks to the entire SG14 group for feedback on initial drafts.

Bibliography

[Bastien, 2018] Bastien, J. (2018). P1152r0 deprecating volatile. Technical Report P1152R0, ISO. Retrieved at wg21.link/P1152R0.

[Bastien, 2019] Bastien, J. (2019). P1152r0 deprecating volatile. Technical Report P1152R4, ISO. Retrieved at wg21.link/P1152R4.

[Craig, 2020] Craig, B. (2020). P2268r0 freestanding roadmap. Retrieved at wg21.link/P2268R0.

[Meredith, 2020] Meredith, A. (2020). P2139 reviewing deprecated facilities of c++20 for c++23. Technical Report P2139, ISO. Retrieved at wg21.link/P2139.

[Ooijen, 2020] Ooijen, W. V. (2020). Compound assignment to volatile must be un-deprecated. https://www.reddit.com/r/cpp/comments/jswz3z/compound_assignment_to_volatile_must_be/.

[von Tessin, 2019] von Tessin, M. (2019). C++ in deeply embedded systems. EmBo++ presentation https://youtu.be/nuwOJ-xUhFU.